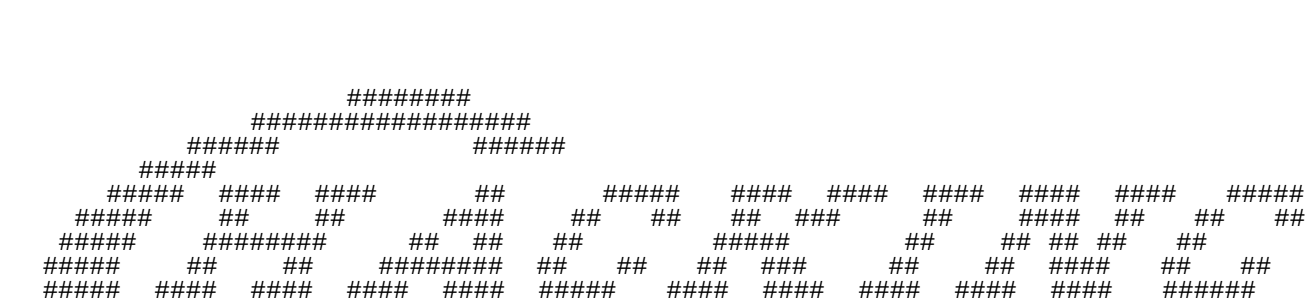


Issue #7
Feb. '94

Editor's Notes:
by Craig Taylor (duck@pembvax1.pembroke.edu)

This net-magazine is still alive after over two years suprising even myself. There should be more to come also!! And it's been late everytime it's been released. :-) I dunno - I'm trying not to make that a tradition but next time I don't think I'm going to set any firm dates so it can't possibly be late.

Lesee, first to address some questions and queries that have been going on over the network:

- The C project that I mentioned a while back is now officially defunct, dead, resting in peace... whatever you want to call it. There may be other individuals working on their own version of an ANSI C compiler but the group that was setup is no longer. Basically, it was a problem of too many programmers, too few managers (and we know programmers *grins* - They all want to do it their way. Incidentilly - my way _was_ the best. ;-))

- The C65 exists but there are very few of them out there and GrapeVine no longer sells them (out of stock I believe). It was only made in small quantities for prototype work. There have been numerous individuals reading comp.sys.cbm that see mention of the C65 and assume that this creature Commodore released. Commodore sold a warehouse and its contents to Grapevine and Grapevine stumbled across them and sold 'em. They are functional but due to the low number of them available I doubt that there will be any C65 targated applications. (Someone prove me wrong, please?)

- Geesh - That stupid article about the 1351 mouse that I kept promising from issue 2 on I've officially declared dead. It may come around later but for now it's dead. It's existed as a promise in one form or another since issue 2 so this message officially kills it.

- Also there was some confusion last time about my wording about not writing any more articles and such (or rather, not as many as I had been). Basically, yes - I'm still going to do C=Hacking - it's just that because of time, frustration, school and jobs that I don't have as much time to devote to researching and writing articles. That's all that I meant to say - some people were concerned that C=Hacking was no longer going to be released...

As always, here's the obligatory begging for authors on any type of software or hardware project that involves the Commodore computers. Please, Please if you or your user group has any information or material concerning the Commodore computers that you think may be appropriate for C=Hacking let me know via e-mail at duck@pembvax1.pembroke.edu. I'm looking for anything from hardware schematic, programming theory to actual programming techniques, programs.

The articles in this issue of C=Hacking I'm especially pleased with. There are two articles concerning the VIC chip and its operations that detail how certain "magical" techniques are performed. Also, Marko Makela has written a very detailed, interesting article on various memory techniques for the C-64. We also have a summary of the ACE programming / operating system and a referance guide on how to write applications for it. I've also included an InterNet guide that I've started writing to show individuals how to use the InterNet to get Commodore-64/128/Vic-20 related material. Included within that is a list of FTP sites containing numerous programs.

==========

Please note that this issue and prior ones are available via anonymous

FTP from ccosun.caltech.edu (among others) under pub/rknop/hacking.mag
and via a mailserver which documentation can be obtained by sending
mail to "duck@pembvax1.pembroke.edu" with a subject line of
"mailserver" and the line "help" in the body of the mesage.

==========

NOTICE: Permission is granted to re-distribute this "net-magazine", in
whole, freely for non-profit use. However, please contact individual
authors for permission to publish or re-distribute articles
seperately. A charge of no greater than 5 US dollars or equivlent may
be charged for library service / diskette costs for this
"net-magazine".

==========

In This Issue:

Commodore Trivia Corner

This section of C=Hacking will contain numerous questions that will test your knowledge of trivia for the Commodore computers. Each issue they'll be answers to the previous issues questions and new questions. How much do you know?

InterNet Resources for the Commodore 64 / 128 V1.0

This article goes into detail about the available resources on the InterNet and is meant to introduce people to the wonderful, wacky world of the InterNet. It covers what the InterNet is, what capabilities it has and how to access those capabilities. In addition, it also includes Howard Herman's latest list of File Transfer sites for the Commodore computers.

Hiding kilobytes

Most Commodore 64 programs do not utilize even nearly all of the 64 kB random access memory space. By default, there are only 44 kilobytes of accessible RAM. This article describes how you can take the hiding 20 kilobytes to use.

FLD - Scrolling the Screen

This article, using a technique described by Pasi Ojala in the last issue of C=Hacking, gives an example of a program using Flexible Line Distance technique.

Tech-tech - more resolution to vertical shift

At one time half of the demos had pictures waving horizontally on the width of the whole screen. This effect is named tech-tech and it is done using character graphics. How exactly and is the same possible with sprites ?

ACE-128/64 PROGRAMMER'S REFERENCE GUIDE (version 0.9, for Release #10)

This article explains the complete system interface for the ACE-128/64 computing environment. It is intended to be used by programmers for developing software to run on top of the ACE kernel. ACE is a program for the Commodore 128 and Commodore 64 that provides a command shell environment that is similar to that of Unix.

==========

Commodore Trivia Corner

by Jim Brain (brain@mail.msen.com)

Everyone who reads this article has had, or presently owns a Commodore computer of some kind. I own more than one, which is not uncommon among Commodore enthusiasts. Well, I bought my first Commodore computer in 1982, a brand new Vic-20 with some game cartridges. I had wanted an Atari, but father knew best and he told the then 11 year old son that computers would shape my life more than any game machine. Well, it is 11 years later and a Computer Engineering Degree earned, and I have spent many a night during that time playing on many models of Commdore equipment. Now, I would like to share the knowledge I have with you in an interesting way.

As you now know a little about me, let us see how much you know about the machines and company that binds us all together. The following is an installment of Commodore Trivia. These questions have been gleaned from books, magazines, persoanl knowledge, work on the machines in questions, and other fellow commodore users happy to share interesting bits of semi- useless knowledge concerning the CBM systems.

This installment consists of two parts, the December 1993 edition complete with answers and the January 1994 edition without answers. Each new issue of Commodore Hacking Magazine will contain more questions, as well as answers to the previous issue's questions. Each new edition is also posted every month on the 12th of the month on the Usenet newsgroup comp.sys.cbm. Winners will be announced on the newsgroup, and prizes may be awarded. For anyone wishing to submit answers from this article, please email your responses with question numbers preceeding each answer to :

brain@msen.com

The answers to this edition will be posted on the 12th of February in comp.sys.cbm with the next edition of questions. Have fun trying to answer the questions and feel free to send me a note with new questions. I can always use them.

[Ed's Note: In addition, the mailserver that I have setup for C=Hacking will make provisions to allow individuals to retrieve the newest set of questions and last month's answers. Currently not implemented, it should be available soon. Details in the next issue.

Also due to C= Hacking being published fairly irregularly and not every month the column here will contain answers to the last issue of C=Hacking's questions and have new questions for the month that it's released in.]

Here are the answers to the 10 questions in Commdore Triva Edition #1 for December 1993. [that were posted on comp.sys.cbm]

Q $000) Commodore started out into computing with the PET series of computers. I am not sure if the first ones had PET emblem, but nonetheless, What does P E T stand for?

A $000) Personal Electronic Transactor
Since the acronym was made up before the expansion, the following are also valid:

Personal Electronic Translator
Peddle's Ego Trip

Q $001) Commodore planned to manufacture a successor series to the successful Commodore 64 home computer. Both were intended to be Business machines. We all know this resulted in the Commodore Plus 4, but what were the machines originally called and what was the difference between the two?

A $001) the 364, which had, among other things, a larger Plus 4 style case that housed the regular keyboard plus a numeric keypad. the 264 turned into the Plus 4, with 64K of RAm. We will never know much more about the 364, since it got scrapped.

Q $002) How much free memory does a Vic-20 have (unexpanded)?

A $002) Oooh! There are many answers for this.
The VIC has 3583 bytes of RAM for BASIC
The VIC has 4096 bytes of RAM for ML
The VIC has 5120 bytes of RAM. 4K of RAM + 1K for Video.

Q $003) What early 80's Commodore software company had a Light Bulb as a company logo?

A $003) Skyles Electric Works.

The Vic-20 came out with a few peripherals I want the model numbers for the :

Q $004) Disk Drive

A $004) VIC-1540 - Same as 1541, only faster serial spped.

Q $005) Cassette Player

A $005) VIC-1530

Q $006) Printer

A $006) VIC 1515, which, by a miscommunication, could only use 7.5" paper. Evidently, someone thought 8.5" meant full width of paper w/ perfs! This printer was quickly supplanted and overtaken by the 1525, which

should own this title in the first place!

Q $007) 16 K Ram Expansion.

A $007) VIC-1111

Q $008) Commodore Introduced 3 printers that used the same printer mechanism.
What are the model numbers.

A $008) MPS 802 (Square Dots, Serial), CBM 1526 (Round Dots, Serial),
PET 4023 (Round Dots, IEEE-488).

Q $009) What is the diferences between the printers in #9

A $009) MPS 802 (Square Dots, Serial), CBM 1526 (Round Dots, Serial),
PET 4023 (Round Dots, IEEE-488).

Here are the questions for Commodore Trivia Edition #2 for January 1994.

Q $00A) What was the Code-Name of the Amiga while in Development?

Q $00B) What is Lord British's Real Name (The creator of the Ultima
Series)?

Q $00C) What is the POKE location and value that will fry an early model
PET?

Q $00D) On the Plus 4 and C-16, the VIC chip was replaced with the TED
chip. What does TED stand for?

Q $00E) Commodore Produced a Letter Quality Printer in North America
(maybe elsewhere) for the Commdore Serial Line. Name it.

Q $00F) What is the version of DOS in the 1541?

Q $010) What is the Version of BASIC in the Plus 4 and the C-16?

Q $011) What are the nicknames of the original three custom Amiga chips?

Q $012) Commodore produced a 64 in a PET case. What is its name and model
number?

Q $013) Commodore sold a 1 megabyte floppy disk drive in a 1541 case.
Give the model number.

Q $014) What does GCR stand for?

Q $015) Commdore produced a drive to accompany the Plus 4 introduction.
Give the model number.

Q $016) What does SID stand for?

Q $017) What does the acronym KERNAL stand for?

Q $018) What version of DOS does the 1571 have?

Q $019) What other two Commdore Disk Drives share the same DOS version
number as the 1571?

Q $01A) How many files will the 1571 hold?

Q $10B) How many files will the 1541 hold?

Q $01C) What did Commodore make right before entering the computer market?

Q $01D) Commodore introduced an ill-fated 4 color plotter. Give the model
number.

Q $01E) Some formats of CP/M write disks using the MFM format. What does
MFM stand for?

Q $01F) On the Commodore 128, the user manual left two commands undocumented.
One works, and the other gives a not-implemented error. Name both
commands and what each one does or does not do.

Some are easy, some are hard, try your hand at it.

Jim Brain
brain@msen.com

==========

InterNet Resources for the Commodore 64 / 128 V1.0
by Craig Taylor (duck@pembvax1.pembroke.edu)

[This article is placed into public domain by the author. Copying encouraged]

The Internet

Let me start this article with a quote by another author that everyone should heed when dealing with the InterNet:

"One warning is perhaps in order---this territory we are entering can become a fantastic time-sink. Hours can slip by, people can come and go, and you'll be locked into Cyberspace. Remember to do your work!

With that, I welcome you, the new user, to The Net."

brendan@cs.widener.edu
 - Author, Zen and the Art of the Internet

What is the InterNet?

What exactly is the InterNet? Imagine if you will, when you were a kid stringing wires between houses in your neighborhood so that you could talk with the kids that lived beside you. You could talk to those beside you but not the ones that lived across town. Now, suppose that you wanted to relay a message to a buddy across town. The only feasible way would be to send a message to the guy next door; then have him send it to the correct person.

This is the basic system of the Internet. Computers connected to other computers that are connected to others. In the above paragrph communication was limited because of geography - how close individuals were. The InterNet system; while geography does play a factor, relies more on how the sites grew up and were established as to how messages will get passed back and forth.

There are also other networks hooked up to the InterNet that provide auxilary services to their local group of computers. One large one is BITNET and UUCP. Various bbs's also carry items from the InterNet such as the BitNet news. In addition, online services such as Genie, Compuserve, and others offer "gate-ways" or ways of getting access to the resources of the InterNet.

Access To The InterNet

Gaining Access to the InterNet There are several ways of gaining access to the InterNet. Your local college may be your best low-cost opportunity. Typically, if you are a student or faculty or staff, you may qualify to have an account that allows you to access all the InterNet facilities described above. If you don't fall into any of these categories your next best bet is an online service such as America Online, Genie, or Compuserve as these all support what is known as an InterNet gateway - allowing you t o access the InterNet through t hem. (At this time, I don't believe Prodigy has an InterNet gateway - if I'm wrong I'm sure I'll get tons of mail. Other online services also exist - I've only listed what I consider the "primary" ones.)

Once you've gotten access to the InterNet you may be asking "Okay, I know what the InterNet _can_ give me - how do I do it?" Unfortunately, because the InterNet is run on differant computer systems this will vary from system to system. Your best bet would be to examine the documentation and help screens associated with the online service or college's facilities. Study them over until you can quote them backwards (well, not quite that much) - Study them over until you understand what they are saying. Also, having someone who is already experienced with the InterNet aid you in your explorations is a great help.

What is E-MAIL?

There are numerous individuals using the InterNet each day. Each is also able to write the other through the use of Electronic Mail or, as it's commonly called "e-mail".

To send a message to me you'd use your mail program (the actual procedure varies depending on what type of machine you use) and tell it to send the message to my user name, "duck" at my site that I login at - (currently going to Pembroke State University) hence "pembvax1.pembroke.edu". So the full address with an "@" sign the computer needs to use to know how to seperate the computer name and the user name is "duck@pembvax1.pembroke.edu".

It's easy to talk to somebody in Mexico, Germany, Australia with this method and it's quicker than the U.S. Postal system (which, on the InterNet you'll see referred to as Snail Mail (or s-mail) due to it's slow delivery time compared to e-mail). Projects, Questions, Answers, Ideas and general chit-chat on how the family is doing, etc can be relayed over the InterNet.

There are also numerous abbreviations and notations that are used in E-Mail. Some of them are:

```
ttyal8r         - Talk to you later
rtfm      - Read the *uckin' manual
imho      - In My Humble Opinion
rotfl     - Rolls on the Floor Laughing
lol       - Laughs Out Loud.
l8r       - Later
;-)       - (winks)
:-)       - (smile)
:-(       - (frowns)
```

There are _many_ _many_ more - you can also find a huge list of the smiley faces (turn your head sideways and look at the ones in parenthesis above) on the InterNet.

You may also hear the phrase "my e-mail bounced". What this means is that your message, much like a bounced check, did not work right and it was returned to your account. Typically this happens because of incorrect addresses, or an incorrect user name.

Email Servers

Another large way of getting information is from individuals running what are E-Mail servers from their accounts or from specific accounts. From Email servers you may request certain files; catalogs of programs that are availble for request; send messages to be distributed to other individuals and automatically subscribe yourself to the mailing list for new items.

The only Email Server specifically designed for the Commodore computers is one ran by the author. It major intent is that of distributing the Commodore Hacking magazine as well as programs that are in the magazine. To get help on how to use it send a message to the author in the following format:

```
  To:   duck@pembvax1.pembroke.edu
  Subj: MAILSERV
  Body of message: HELP
```

This specific mailserver is ran twice a day so you should get your reply within approximately 12 hours. Please be sure to have a subject line of "MAILSERV".

If anyone knows of any other Email Servers existing for the Commodore computers please let the author know.

NewsGroups

One of the primary purposes of the InterNet is for educational research and discussion. For this purpose, there are currently over 2000 newsgroups established dealing with a wide range of social, politicial, science, computer and educational topics. Some of these range to inane, whimsical, to practical and useful.

Two of these for the Commodore 64/128 line of computers are:

```
        comp.sys.cbm
        comp.binaries.cbm
```

The names for the newsgroups start with a short abbreviation such as "comp" for computers, "sci" for science, "bio" for biology, etc... The second group of letters stand for the type of newsgroup "sys for system, binaries for binaries etc..." while the third describes it better - "cbm" in this case for Commodore Business Machines.

The newsgroup, Comp.Sys.Cbm supports discussion about anything under the sun involving the Commodore 8 bit line of computers (and lately, even talking about the old old ancient calculators that Commodore mae that might not have even been 8 bit). Comp.Binaries.Cbm allows programs to be "posted" or made available to everyone who wishes them. There are programs available that will let you take the "encrypted" text-only version of the program that you see on the screen and convert them into the correct binary p rogram.

Basically the rules for newsgroups are: 1) Enjoy yourself, 2) Don't harass others and 3) Try to stay on topic. Newsgroups are read by many many people - typically you'll get a response to an inquiry within only an hour or so - sometimes even sooner. But because they're read by so many people chatter or "babble" as it's known, is also discouraged. Don't hesitate to post any questions, concerns or comments but make sure in each message that you post that you have a reason to post.

So What's Out There?

So why should you be interested in the Internet? Imagine, if you will, being able to ask questions to numerous individuals, download the latest in shareware and public-domain software, know the "rumours" and topics before they exist all for free? (Or at least, only for what your "hookup" method charges - see Gaining Access to the InterNet latter). That's what's out on the Internet. Any question you have - there is sure to be an answer for - any software you're looking for you stand an extremely good chance of finding something along the lines of your needs.

The major benefit of the Internet as I see it consists of the continued support for the Commodore computers. Because all these differant means of obtaining information are not sponsored by any one specific company or individual the Commodore 8-bit line of computers are guranteed continued support over the Internet. In addition, because the Internet strongly frowns upon Commercial advertising you won't find numerous ads or any other material urging you to "buy this, buy that" like you will on some other serv ices.

FTP Sites

FTP stands for File Transfer Protocols and is a method of obtaining programs that are stored on another system's computers. There are numerous FTP sites out there in InterNet land - one of the best currently available for the Commodore computers is that of R.Knop's site at ccosun.caltech.edu.

[The following is a list of FTP sites for the Commodore 64 and 128 computes and is currenty maintained by Howard Herman (h.herman@GEnie.geis.com) and is used with his permission. He usually posts an updated list to comp.sys.cbm newsgroup every month or so.]

This is the list of FTP sites containing software and programs specific to the Commodore 64 and 128 computers.

I will try to keep this list as current and accurate as possible, so that it can be a useful resource for users of the newsgroup.

PLEASE cooperate and send E-mail to me with any corrections and updates. If a site has closed or no longer carries CBM software, let me know and it will be deleted. If you uncover a site not listed, tell me so that it can be added.

To use this list on a UNIX system, just type 'ftp <sitename>', where <sitename> is any of the sites listed below. Use 'anonymous' as your login, and your E-mail address for the password. You can change and list directories with 'cd' and 'dir', respectively, and download files to your system using 'get'. Be sure to specify either 'binary' if you are getting a program, or 'ascii' for a text file before you begin the download.

In addition to the sites listed below there are hundreds of other FTP sites on INet with interesting files covering every topic imaginable. Take some time to seek out and explore these too.

Enjoy!

```
Host sol.cs.ruu.nl   (131.211.80.17)
Last updated 00:39  4 Sep 1993
    Location: /pub/MIDI/PROGRAMS
      DIRECTORY rwxr-xr-x      1024  Aug 26 09:58   C64
    Location: /pub/MIDI/DOC
      FILE      rw-r--r--     40183  Jan 19  1993   C64midi-interface.txt

Host uceng.uc.edu   (129.137.33.1)
Last updated 04:38  6 Sep 1993
    Location: /pub/wuarchive/systems/cpm/c128
      FILE      rw-r--r--     24576  Nov  6  1986   c128-mex.com
```

```
    Location: /pub/wuarchive/systems/cpm/c64
      FILE      rw-r--r--      1615  Mar 14  1984   c64-cpm.msg
      FILE      rw-r--r--      1536  Feb  9  1985   c64modem.com
      FILE      rw-r--r--      4199  Feb 10  1984   c64modem.doc
      FILE      rw-r--r--     19200  Feb  9  1985   md730c64.com
      FILE      rw-r--r--      2192  Oct  1  1984   md730c64.doc
    Location: /pub/wuarchive/doc/misc/if-archive/infocom/tools
      FILE      r--r--r--      5798  Aug  5 14:42   c64todat.tar.Z

Host aix370.rrz.uni-koeln.de   (134.95.80.1)
    Location: /.disk2/usenet/comp.archives/auto/comp.sys.cbm
      DIRECTORY rwxrwxr-x       384  comp.sys.cbm

Host ftp.csv.warwick.ac.uk     (137.205.192.5)
Last updated 00:00 18 Jan 1994
    Location: /pub/c64
      FILE      rw-r--r--       909 Jan  8 19:52 C64progs.doc
      FILE      rw-r--r--     19558 Jan  8 19:49 backgamm.sfx
      FILE      rw-r--r--     21384 Jan  8 19:49 chequebo.sfx
      FILE      rw-r--r--     11449 Jan  8 19:49 countdow.sfx
      FILE      rw-r--r--     18136 Jan  8 19:49 draughts.sfx
      FILE      rw-r--r--      5011 Jan  8 19:49 loader.sfx
      FILE      rw-r--r--     17423 Jan  8 19:49 whitewas.sfx
Descriptions:
============
backgamm - Backgammon board game
chequebo - Cheque Book Organiser, written in basic with UK pound sign as
           currency, but could be changed to suit another.
countdow - LOAD"count example",8,1 and watch the countdown during loading.
draughts - Draughts board game.
loader   - Press RESTORE key and MENU on disk will be automatically re-loaded.
whitewas - Colour squares board game.
    Location: /tmp/c64
     >Temporary files stored here.  If /tmp directory not found, try
     >again at another time.  /tmp directory not always available.

Host clover.csv.warwick.ac.uk   (137.205.192.6)
Last updated 13:29 27 Sep 1993
    Location: /pub/c64
      FILE      rw-r--r--       812  c64progs.doc
      FILE      rw-r--r--     73696  c64progs.sfx

Host nexus.yorku.ca   (130.63.9.66)
Last updated 00:00 21 Dec 1993
    Location: /pub/Internet-info
      FILE      rw-r--r--      6308  commodore.ftp
     >An older version of this listing.

Host rigel.acs.oakland.edu   (141.210.10.117)
Last updated 01:42  3 Sep 1993
    Location: /pub2/cpm
      DIRECTORY rwxr-xr-x      1536  Jun  4  1992   c128
      DIRECTORY rwxr-xr-x       512  c64
    Location: /pub2/cpm/c64
      FILE      rw-r--r--      1615  Mar 14  1984   c64-cpm.msg
      FILE      rw-r--r--      1536  Feb  9  1985   c64modem.com
      FILE      rw-r--r--      4199  Feb 10  1984   c64modem.doc
      FILE      rw-r--r--     19200  Feb  9  1985   md730c64.com
      FILE      rw-r--r--      2192  Oct  2  1984   md730c64.doc

Host oak.oakland.edu
Last updated 00:00 18 Dec 1993
    Location: /pub2/cpm
     >For CP/M software, most all of which will run on the C128.

Host src.doc.ic.ac.uk   (146.169.2.1)
    Location: /usenet/comp.archives/auto
      DIRECTORY rwxr-xr-x       512  comp.sys.cbm
    Location: /usenet/comp.archives
      DIRECTORY rwxr-xr-x       512  commodore-64-128
      DIRECTORY rwxr-xr-x       512  May  3  1991   c64
    Location: /media/visual/collections/funet-pics/jpeg/games
      DIRECTORY rwxr-xr-x       512  Mar 20 05:48   c64
    Location: /media/visual/collections/funet-pics/jpeg/comp/games
      DIRECTORY rwxr-xr-x       512  May  6 03:55   c64

Host tupac-amaru.informatik.rwth-aachen.de   (137.226.112.31)
Last updated 04:59  7 Oct 1992
    Location: /pub/rz.archiv/simtel/cpm
      DIRECTORY rwxr-xr-x       512  c64
      DIRECTORY rwxr-xr-x       512  Sep 21 20:56   c128
```

```
Host wuarchive.wustl.edu   (128.252.135.4)
Last updated 02:40 23 May 1993
    Location: /systems/amiga/incoming/misc
      FILE      rw-rw-r--     21815  Jan 23 14:26   C64View.lha
      FILE      rw-rw-r--       120  Jan 23 14:26   C64View.readme
    Location: /mirrors/cpm
      DIRECTORY rwxr-xr-x       512  c64
      DIRECTORY rwxr-xr-x      1536  Nov 22  1992   c128

Host watsun.cc.columbia.edu   (128.59.39.2)
Last updated 02:07  8 Sep 1993
    Location: /kermit2/old
      DIRECTORY rwxrwxr-x      1024  Jul 12 18:30   c64
    Location: /kermit/bin

Host cs.columbia.edu   (128.59.1.2)
Last updated 01:29 12 Sep 1993
    Location: /archives/mirror1/kermit
      FILE      rw-rw-r--     15016  Aug 24  1988   c644th.prg.gz
      FILE      rw-rw-r--       733  Sep 29  1992   c64help.txt.gz
      FILE      rw-rw-r--      6095  Sep 29  1992   c64ker1660.sda.gz
      FILE      rw-rw-r--      5904  Sep 29  1992   c64kerfast.sda.gz
      FILE      rw-rw-r--     26484  Sep 29  1992   c64kerv22a.sda.gz
      FILE      rw-rw-r--     42552  Sep 29  1992   c64kerv22b.sda.gz
      FILE      rw-rw-r--     31982  Sep 29  1992   c64slkv22s.sda.gz

Host plaza.aarnet.edu.au   (139.130.4.6)
Last updated 00:00 28 Dec 1993
    Location: /pub/kermit/c
      FILE      r--r--r--      3073  Aug 16  1988   c64boot.bas
      FILE      r--r--r--      1547  Aug 16  1988   c64boot.c
      FILE      r--r--r--      1151  Aug 16  1988   c64boot.clu
      FILE      r--r--r--      3002  Aug 16  1988   c64boot.for
      FILE      r--r--r--      3315  Aug 16  1988   c64boot.sim
     >There are more Kermit files which are not listed.  Be sure to
     >get the complete set of C64/128 Kermit files.

Host flubber.cs.umd.edu   (128.8.128.99)
Last updated 00:00 03 Jan 1994
    Location: /rec/newballistic
      FILE      rw-r--r--      8576  Mar 23 21:21   balistic.c64
--------------------------------

Host f.ms.uky.edu (128.163.128.6)
Last updated 00:00 28 Dec 1993
    Location: /archive/c64.zip

Host ftp.funet.fi (128.214.6.100)
Last updated 06:11 22 Mar 1993
    Location: /pub/pics/jpeg/games
      DIRECTORY rwxrwxr-x       512  Mar 20 02:07   c64
    Location: /pub/misc
      DIRECTORY rwxrwxr-x       512  Mar 13 23:30   c64
    Location: /pub/kermit
      DIRECTORY rwxrwxr-x      1024  Jan 13  1992   c64
    Location: /pub/amiga/audio/misc/sid-tunes
      FILE      rw-rw-r--    671490  Jun 18  1992   C64MusicShow-1.lha
      FILE      rw-rw-r--    316521  Jun 18  1992   C64MusicShow-2.lha
              /pub/cbm

Host nic.switch.ch   (130.59.1.40)
Last updated 00:39 31 Aug 1993
    Location: /mirror/kermit/bin

Host gmdzi.gmd.de   (129.26.8.90)
Last updated 01:08  1 Aug 1993
    Location: /if-archive/infocom/tools
      FILE      rw-rw-r--      5668  Apr 27 15:00   c64.to.dat

Host micros.hensa.ac.uk   (148.88.8.84)
    Location /kermit
      DIRECTORY rwxr-x---      1024  Nov 11 09:20   c64

Host wilbur.stanford.edu   (36.14.0.36)
    Location /pub/emulators
      DIRECTORY rwxr-xr-x       512  Jun 30 00:57   c64

Host syrinx.umd.edu (128.8.2.114)
Last updated 00:00 28 Dec 1993
    Location: /rush/c64-sounds
```

```
Host tolsun.oulu.fi (130.231.96.16)
Last updated 01:53  6 Sep 1993
    Location: /pub
      DIRECTORY rwxr-xr-x      1024  Jul 15  1990   c64
    Location: /incoming
      DIRECTORY ---------      1024  Jun 20  1992   c64
        /pub/amiga/4/c64trans.zoo
        /pub/c64
     >Uploading to /pub/c64 is disabled because of lack of disk space.
     >However, for downloading it is still fully accessible.
     >Currently there is no administration for /pub/c64.
     >/pub/amiga is active, though.

Host ccosun.caltech.edu (131.215.139.2)
Last updated 00:00 31 JAN 1994
    Location: /pub/rknop
    Location: /pub/rknop/misc
     > 64/128 programs can be found within directories according to
     > function.  When searching, be sure to check related directories.

Host ucsd.edu (128.54.16.1)
Last updated 04:46  6 Sep 1993
    Location: /midi/software
      DIRECTORY rwxr-xr-x       512  Jan 27  1992   c64

Host cs.dal.ca   (129.173.4.5)
Last updated 01:36 12 Sep 1993
    Location: /comp.archives
      DIRECTORY rwxrwxr-x      3584  Apr  7 04:05   c64
                pub/comp.archives/comp.sys.cbm

Host ccnga.uwaterloo.ca
Last updated 00:00 01 Jan 1994
    Location: /pub/cbm/vbm110.uua
     > For VBM Bitmap Viewer version 1.10

Host bert.psyc.upei.ca
Last updated 00:00 31 Jan 1994
    Location: /pub
     > All the releases of the major demo parties of '93

--------------------------------
Send all info regarding changes/additions/corrections [to the FTP list] to:

72560.3467@CompuServe.COM   or:     h.herman1@GEnie.geis.COM
================================================================================
```

Hiding kilobytes
by Marko M"akel"a <Marko.Makela@Helsinki.FI>

Most Commodore 64 programs do not utilize even nearly all of the 64 kB random access memory space. By default, there are only 44 kilobytes of accessible RAM. This article describes how you can take the hiding 20 kilobytes to use.

Memory management

The Commodore 64 has access to more memory than its processor can directly handle. This is possible by banking the memory. There are five user configurable inputs that affect the banking. Three of them can be controlled by program, and the rest two serve as control lines on the memory expansion port.

The 6510 MPU has an integrated I/O port with six I/O lines. This port is accessed through the memory locations 0 and 1. The location 0 is the Data Direction Register for the Peripheral data Register, which is mapped to the other location. When a bit in the DDR is set, the corresponding PR bit controls the state of a corresponding Peripheral line as an output. When it is clear, the state of the Peripheral line is reflected by the Peripheral register. The Peripheral lines are numbered from 0 to 5, and they are mapped to the DDR and PR bits 0 - 5, respectively. The 8502 processor, which is used in the Commodore 128, has seven Peripheral lines in its I/O port. The seventh line is connected to the Caps lock or ASC/CC key.

The I/O lines have the following functions:

Direction	Line	Function
out	P5	Cassette motor control. (0 = motor spins)

```
        in      P4   Cassette sense. (0 = PLAY button depressed)
        out     P3   Cassette write data.
        out     P2   CHAREN
        out     P1   HIRAM
        out     P0   LORAM
```

The default value of the DDR register is $2F, so all lines except Cassette sense are outputs. The default PR value is $37 (Datassette motor stopped, and all three memory management lines high).

Like most chips in the Commodore 64, the 6510 MPU uses the NMOS (N-channel metal oxide semiconductor) technology. The NMOS switches produce strong logical '0' levels, but weak '1' levels. The opposite is the PMOS (P-channel metal oxide semiconductor) technology, which cannot pull strong signals low, but is able to drive them high. The CMOS technology (complementary metal oxide semiconductor), which combines these two technologies, is able to drive both logical levels.

Because most integrated circuits in the C64 use the NMOS technology, all hardware lines that are not outputs are driven to +5 volts with a weak current. This is usually accomplished by pull-up resistors, large resistances between the hardware lines and the +5 volt power supply line. The resistors can be inside a chip or on the printed circuit board. This allows any NMOS or CMOS chip to drive the line to the desired state (low or high voltage level).

The difference between an input and an output is that an output uses more current to drive the signal to the desired level. An input and an output outputting logical '1' are equivalent for any other inputting chip. But if a chip is trying to drive a signal to ground level, it needs more current to sink an output than an input. You can even use outputs as inputs, i.e. read them in your program.

> You can use this feature to distinquish between the left shift and the shift lock keys, although they are connected to same hardware lines. The shift lock key has smaller resistance than the left shift. If you make both CIA 1 ports to outputs (write $FF to $DC03 and $DC01) prior reading the left shift key, only shift lock can change the values you read from CIA 1 port B ($DC01).)

So, if you turn any memory management line to input, the external pull-up resistors will make it look like it is outputting logical '1'. This is actually why the computer always switches the ROMs in upon startup: Pulling the -RESET line low resets all Peripheral lines to inputs, thus driving all three processor-driven memory management lines high.

The two remaining memory management lines are -EXROM and -GAME on the cartridge port. Each line has a pull-up resistor, so the lines are '1' by default. (In the Commodore 128, you can set the state of these two lines prior to selecting the C64 mode, provided that you write the mode switch routine yourself.)

Even though the memory banking has been implemented with a 82S100 Programmable _Logic_ Array, there is only one control line that seems to behave logically at first sight, the -CHAREN line. It is mostly used to choose between I/O address space and the character generator ROM. The following memory map introduces the oddities of -CHAREN and the other memory management lines. It is based on the memory maps in the Commodore 64 Programmer's Reference Guide, pp. 263 - 267, and some errors and inaccuracies have been corrected.

The leftmost column of the table contains addresses in hexadecimal notation. The columns aside it introduce all possible memory configurations. The default mode is on the left, and the absolutely most rarely used Ultimax game console configuration is on the right. (There have been at least two Ultimax cartridges on the market.) Each memory configuration column has one or more four-digit binary numbers as a title. The bits, from left to right, represent the state of the -LORAM, -HIRAM, -GAME and -EXROM lines, respectively. The bits whose state does not matter are marked with "x". For instance, when the Ultimax video game configuration is active (the -GAME line is shorted to ground, -EXROM kept high), the -LORAM and -HIRAM lines have no effect.

```
      default                                                        Ultimax
        1111    101x    1000    011x    001x    1110    0100    1100    xx01
                                00x0
10000
--------------------------------------------------------------------------------
```

F000 E000	Kernal	RAM	RAM	Kernal	RAM	Kernal	Kernal	Kernal	ROMH(*
D000	IO/C	IO/C	IO/RAM	IO/C	RAM	IO/C	IO/C	IO/C	I/O
C000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
B000 A000	BASIC	RAM	RAM	RAM	RAM	BASIC	ROMH	ROMH	-
9000 8000	RAM	RAM	RAM	RAM	RAM	ROML	RAM	ROML	ROML(*
7000									
6000 5000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
4000									
3000									
2000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
1000									
0000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM

*) Internal memory does not respond to write accesses to these areas.

Legend:	Kernal	E000-FFFF	Kernal ROM.
	IO/C	D000-DFFF	I/O address space or Character generator ROM, selected by -CHAREN. If the CHAREN bit is clear, the character generator ROM is chosen. If it is set, the I/O chips are accessible.
	IO/RAM	D000-DFFF	I/O address space or RAM, selected by -CHAREN. If the CHAREN bit is clear, the character generator ROM is chosen. If it is set, the internal RAM is accessible.
	I/O	D000-DFFF	I/O address space. The -CHAREN line has no effect.
	BASIC	A000-BFFF	BASIC ROM.
	ROMH	A000-BFFF or E000-FFFF	External ROM with the -ROMH line connected to its -CS line.
	ROML	8000-9FFF	External ROM with the -ROML line connected to its -CS line.
	RAM	various ranges	Commodore 64's internal RAM.
	-	1000-7FFF and A000-CFFF	Open address space. The Commodore 64's memory chips do not detect any memory accesses to this area except the VIC-II's DMA and memory refreshes.

NOTE: Whenever the processor tries to write to any ROM area (Kernal, BASIC, CHAROM, ROML, ROMH), the data will get "through the ROM" to the C64's internal RAM.

For this reason, you can easily copy data from ROM to RAM, without any bank switching. But implementing external memory expansions without DMA is very hard, as you have to use the Ultimax memory configuration, or the data will be written both to internal and external RAM.

However, this is not true for the Ultimax game configuration. In that mode, the internal RAM ignores all memory accesses outside the area $0000-$0FFF, unless they are performed by the VIC, and you can write to external memory at $1000-$CFFF and $E000-$FFFF, if any, without changing the contents of the internal RAM.

A note concerning the I/O area

The I/O area is divided as follows:

Address range	Owner
D000-D3FF	MOS 6567/6569 VIC-II Video Interface Controller
D400-D7FF	MOS 6581 SID Sound Interface Device
D800-DBFF	Color RAM (only lower nybbles are connected)
DC00-DCFF	MOS 6526 CIA Complex Interface Adapter #1
DD00-DDFF	MOS 6526 CIA Complex Interface Adapter #2
DE00-DEFF	User expansion #1 (-I/O1 on Expansion Port)
DF00-DFFF	User expansion #2 (-I/O2 on Expansion Port)

As you can see, the address ranges for the chips are much larger than required. Because of this, you can access the chips through multiple memory areas. The VIC-II appears in its window every $40 addresses. For instance, the addresses $D040 and $D080 are both mapped to the Sprite 0 X co-ordinate register. The SID has one register selection line less, thus it appears at every $20 bytes. The CIA chips have only 16 registers, so there are 16 copies of each in their memory area.

However, you should not use other addresses than those specified by Commodore. For instance, the Commodore 128 mapped its additional I/O chips to this same memory area, and the SID responds only to the addresses D400-D4FF, also when in C64 mode. And the Commodore 65, which unfortunately did not make its way to the market, could narrow the memory window reserved for the MOS 6569/6567 VIC-II (or CSG 4567 VIC-III in that machine).

The video chip

The MOS 6567/6569 VIC-II Video Interface Controller has access to only 16 kilobytes at a time. To enable the VIC-II to access the whole 64 kB memory space, the main memory is divided to four banks of 16 kB each. The lines PA0 and PA1 of the second CIA are the inverse of the virtual VIC-II address lines VA14 and VA15, respectively. To select a VIC-II bank other than the default, you must program the CIA lines to output the desired bit pair. For instance, the following code selects the memory area $4000-$7FFF (bank 1) for the video controller:

```
LDA $DD02 ; Data Direction Register A
ORA #$03  ; Set pins PA0 and PA1 to outputs
STA $DD02
LDA $DD00
AND #$FC  ; Mask the lowmost bit pair off
ORA #$02  ; Select VIC-II bank 1 (the inverse of binary 01 is 10)
STA $DD00
```

Why should you set the pins to outputs? Hardware RESET resets all I/O lines to inputs, and thanks to the CIA's internal pull-up resistors, the inputs actually output logical high voltage level. So, upon -RESET, the video bank 0 is selected automatically, and older Kernals could leave it uninitialized.

Note that the VIC-II always fetches its information from the internal RAM, totally ignoring the memory configuration lines. There is only one exception to this rule: The character generator ROM. Unless the Ultimax mode is selected, VIC-II "sees" character generator ROM in the memory areas 1000-1FFF and 9000-9FFF. If the Ultimax configuration is active, the VIC-II will fetch all data from the internal RAM.

An application: Making an operating system extension

If you are making a memory resident program and want to make it as invisible to the system as possible, probably the best method is keeping most of your code under the I/O area (in the RAM at $D000-$DFFF). This area is very safe, since programs utilizing it are rare, since they are very difficult to implement and to debug. You

need only a short routine in the normally visible RAM that pushes the current value of the processor's I/O register $01 on stack, switches RAM on to $D000-$DFFF and jumps to this area. Returning from the $D000-$DFFF area is possible even without any routine in the normally visible RAM area. Just write an RTS or an RTI to an I/O register and return through it.

But what if your program needs to use I/O? And how can you write the return instruction to an I/O register while the I/O area is switched off? You need a swap area for your program in normally visible memory. The first thing your routine at $D000-$DFFF does is copying the I/O routines (or the whole program) to normally visible memory, swapping the bytes. For instance, if your I/O routines are initially being stored at $D200-$D3FF, exchange the bytes in $D200-$D3FF with the contents of $C000-$C1FF. Now you can call the I/O routines from your routine at $D000-$DFFF, and the I/O routines can switch the I/O area temporarily on to access the I/O chips. And right before exiting your program at $D000-$DFFF swaps the old contents of that I/O routine area in, e.g. exchanges the memory areas $D200-$D3FF and $C000-$C1FF again.

What I/O registers can you use for the return instruction? There are basically two alternatives: 8-bit VIC sprite registers or either CIA's serial port register. The VIC registers are easiest to use, as they act precisely like memory places: you can easily write the desired value to a register. But the CIA register is usually better, as changing the VIC registers might change the screen layout.

However, also the SP register has some drawbacks: If the machine's CNT1 and CNT2 lines are connected to a frequency source, you must stop either CIA's Timer A to use the SP register method. Normally the 1st CIA's Timer A is the main hardware interrupt source. And if you use the Kernal's RS232, you cannot stop the 2nd CIA's Timer A either. Also, if you don't want to lose any CIA interrupts, you might want to know that executing the RTS or RTI at SP register has the side effect of reading the Interrupt Control Register, thus acknowledging an interrupt that might have been waiting.

If you can't use either method, you can use either CIA's ToD seconds or minutes or ToD alarm time for storing an RTI. Or, if you don't want to alter any registers, use the VIC-II's light pen register. Before exiting, wait for appropriate raster line and trig the light pen latch with CIA1's PB4 bit. However, this method assumes that the control port 1's button/light pen line remains up for that frame. After trigging the light pen, causing the light pen Y co-ordinate register ($D014) to be $40 or $60, you have more than half a frame time to restore the state of the I/O chips and return through the register.

You can also use the SID to store an RTI or RTS command. How is this possible, you might ask. After all, the chip consists of read only or write only registers. However, there are two registers that can be controlled by program, the envelope generator and oscillator outputs of the third voice. This method requires you to change the frequency of voice 3 and to select a waveform for it. This will affect on the voice output by turning the voice 3 off, but who would keep the voice 3 producing a tone while calling an operating system routine?

Also keep in mind that the user could press RESTORE while the Kernal ROM and I/O areas are disabled. You could write your own non-maskable interrupt (NMI) handler (using the NMI vector at $FFFA), but a fast loader that uses very tight timing would still stop working if the user pressed RESTORE in the middle of a data block transfer. So, to make a robust program, you have to disable NMI interrupts. But how is this possible? They are Non-Maskable after all. The NMI interrupt is edge-sensitive, the processor jumps to NMI handler only when the -NMI line drops from +5V to ground. To disable the interrupt, simply cause an NMI with CIA2's timer, but don't read the Interrupt Control register. If you need to read $DD0D in your program, you must add a NMI handler just in case the user presses RESTORE. And don't forget to raise the -NMI line upon exiting the program. Otherwise the RESTORE key does not work until the user issues a -RESET or reads the ICR register explicitly. (The Kernal does not read $DD0D, unless it is handling an interrupt.) This can be done automatically by the two following SP register examples due to one of the 6510's undocumented features (refer to the descriptions of RTS and RTI below).

```
        ; Returning via VIC sprite 7 X co-ordinate register

        Initialization:   ; This is executed when I/O is switched on
                LDA #$60
                STA $D015 ; Write RTS to VIC register $15.
```

```
Exiting:          ; NOTE: This procedure must start at VIC register
                  ; $12. You have multiple alternatives, as the VIC
                  ; appears in memory at $D000+$40*n, where $0<=n<=$F.

        PLA       ; Pull the saved 6510 I/O register state from stack
        STA $01   ; Restore original memory bank configuration
                  ; Now the processor fetches the RTS command from the
                  ; VIC register $15.


; Returning via CIA 2's ToD or ToD alarm seconds register

Initialization:   ; This is executed when I/O is switched on
        LDA #$40
        STA $DD08 ; Set ToD tenths of seconds
                  ; (clear it so that the seconds register
                  ; would not overflow)
                  ; If ToD alarm register is selected, this
                  ; instruction will be unnecessary.
        STA $DD09 ; Set ToD seconds
        LDA $DD0B ; Read ToD hours (freeze ToD display)

Exiting:          ; NOTE: This procedure must start at CIA 2 register
                  ; $6. As the CIA 2 appears in memory at $DD00+$10*n,
                  ; where 0<=n<=$F, you have sixteen alternatives.
        PLA
        STA $01   ; Restore original memory bank configuration
                  ; Now the processor fetches the RTS command from
                  ; the CIA 2 register $9.


; Returning via CIA 2's SP register (assuming that CNT2 is stable)

Initialization:   ; This is executed when I/O is switched on
        LDA $DD0E ; CIA 2's Control Register A
        AND #$BF  ; Set Serial Port to input
        STA $DD0E ; (make the SP register to act as a memory place)
        LDA #$60
        STA $DD0C ; Write RTS to CIA 2 register $C.

Exiting:          ; NOTE: This procedure must start at CIA 2 register
                  ; $9. As the CIA 2 appears in memory at $DD00+$10*n,
                  ; where 0<=n<=$F, you have sixteen alternatives.
        PLA
        STA $01   ; Restore original memory bank configuration
                  ; Now the processor fetches the RTS command from
                  ; the CIA 2 register $C.


; Returning via CIA 2's SP register, stopping the Timer A
; and forcing SP2 and CNT2 to output

Initialization:   ; This is executed when I/O is switched on
        LDA $DD0E ; CIA 2's Control Register A
        AND #$FE  ; Stop Timer A
        ORA #$40  ; Set Serial Port to output
        STA $DD0E ; (make the SP register to act as a memory place)
        LDA #$60
        STA $DD0C ; Write RTS to CIA register $C.

Exiting:          ; NOTE: This procedure must start at CIA 2 register
                  ; $9. As the CIA 2 appears in memory at $DD00+$10*n,
                  ; where 0<=n<=$F, you have sixteen alternatives.
        PLA
        STA $01   ; Restore original memory bank configuration
                  ; Now the processor fetches the RTS command from
                  ; the CIA 2 register $C.

; Returning via SID oscillator 3 output register

Initialization:   ; This is executed when I/O is switched on
        LDA #$20  ; Select sawtooth waveform
        STA $D412 ; but do not enable the sound
        LDY #$00  ; Select frequency
        STY $D40E ; (system clock)/$FF00,
        LDA #$FF  ; causing the OSC3 output to increment by one
        STY $D40F ; every $10000/$FF00 cycles.

        LDA #$0E
        LDX #$60
```

```
                BIT $D41B ; Wait for the oscillator 3 output
                BMI *-3   ; to be in the range
                BVS *-5   ; $00-$3F.
                BIT $D41B ; Wait for the oscillator 3 output
                BVC *-3   ; to be at least $40.

                STA $D40F ; Slow down the frequency to (system clock)/$0E00.
                CPX $D41B ; Wait for the oscillator 3
                BNE *-3   ; output to reach $60 (RTS)

                STY $D40F ; Reset the frequency of voice 3
                          ; (stop the OSC3 register from increasing)

        Exiting:          ; NOTE: This procedure must start at SID register
                          ; $18. As the SID appears in memory at $D400+$20*n,
                          ; where 0<=n<=$20, you have thirty-two alternatives.
                          ; However, in C128 there are only eight alternatives,
                          ; as the SID is only at $D400-$D4FF.

                PLA
                STA $01   ; Restore original memory bank configuration
                          ; Now the processor fetches the RTS command from
                          ; the SID register $1B.
```

For instance, if you want to make a highly compatible fast loader, make the ILOAD vector ($0330) point to the beginning of the stack area. Remember that the BASIC interpreter uses the first bytes of stack while converting numbers to text. A good address is $0120. Robust programs practically never use so much stack that it could corrupt this routine. Usually only crunched programs (demos and alike) use all stack in the decompression phase. They also make use of the $D000-$DFFF area.

This stack routine will jump to your routine at $D000-$DFFF, as described above. For performance's sake, copy the whole byte transfer loop to the swap area, e.g. $C000-$C1FF, and call that subroutine after doing the preliminary work. But what about files that load over $C000-$C1FF? Wouldn't that destroy the transfer loop and jam the machine? Not necessarily. If you copy those bytes to your swap area at $D000-$DFFF, they will be loaded properly, as your program restores the original $C000-$C1FF area.

If you want to make your program user-friendly, put a vector initialization routine to the stack area as well, so that the user can restore the fast loader by issuing a SYS command, rather than loading it each time he has pressed RESET.

An example: A "hello world" program

To help you in getting started, I have written a small example program that echoes the famous message "hello, world!" to standard output (normally screen) using the Kernal's CHROUT subroutine. After the initialization routine has been run, the program can be started by commanding SYS 300. I used the Commodore 128's machine language monitor to put it up, but it was still pretty difficult to debug the program. Here it is in uuencoded format:

```
begin 644 hello
M`0@+",D'GC(P-C$```!XI0%(*?B%`:(,O3`(G2P!RA#WHHN]/`B=8]W*T/=H
MA0%88*4!JBGX"01XA0%,I-WF`:*!C@W=H@".!=WHC@3=HMV.#MVB0(X,W<8!
M8*4!2`D#A0&@#+DSP"#2_X@0]VB%`6`A1$Q23U<@+$],3$5(BDBM^O](K?O_
M2*D6C?K_J<"-^_\@W-T@`,!HC?O_:(WZ_R`=P"#<W6BHJ0!(NOX"`=`#_@,!
5A`&@/[X`P+EDW9D`P(J99-V($/!@
`
end
```

In order to fully understand the operation of this program, you need to know how the instructions RTI, RTS and PHA work. There is some work going on to reverse engineer the NMOS 6502 microprocessor to large extent, and it is now known for most instructions what memory places they access during their execution and for what purpose. The internal procedures haven't been described in detail yet, but these descriptions should be easier to read anyway.

For curiosity, I quote here the description of all instructions that use the stack. The descriptions of internal operations are yet inaccurate, but the memory accesses have been verified with an oscilloscope. I will mail copies the whole document upon request. When

finished, the document will be put on an FTP site.

JSR

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	fetch address's low byte to latch, increment PC
3	$0100,S	R	
4	$0100,S	W	push PCH on stack, decrement S
5	$0100,S	W	push PCL on stack, decrement S
6	PC	R	copy latch to PCL, fetch address's high byte to latch, copy latch to PCH

RTS

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	read next instruction byte (and throw it away), increment PC
3	$0100,S	R	increment S
4	$0100,S	R	pull PCL from stack, increment S
5	$0100,S	R	pull PCH from stack
6	PC	R	increment PC

BRK

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	read next instruction byte (and throw it away), increment PC
3	$0100,S	W	push PCH on stack, decrement S
4	$0100,S	W	push PCL on stack, decrement S
5	$0100,S	W	push P on stack (with B flag set), decrement S, set I flag
6	$FFFE	R	fetch PCL
7	$FFFF	R	fetch PCH

RTI

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	read next instruction byte (and throw it away), increment PC
3	$0100,S	R	increment S
4	$0100,S	R	pull P from stack, increment S
5	$0100,S	R	pull PCL from stack, increment S
6	$0100,S	R	pull PCH from stack

PHA, PHP

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	read next instruction byte (and throw it away), increment PC
3	$0100,S	W	push register on stack, decrement S

PLA, PLP

#	address	R/W	description
1	PC	R	fetch opcode, increment PC
2	PC	R	read next instruction byte (and throw it away), increment PC
3	$0100,S	R	increment S
4	$0100,S	R	pull register from stack

The example program consists of three parts. The first part transfers the other parts to appropriate memory areas. The second part is located in stack area (300-312), and it invokes the third part, the main module.

The loader part ($0801-$08C7) is as follows:

```
      1993 SYS2061

      080D SEI          Disable interrupts.
      080E LDA $01
      0810 PHA          Store the state of the processor's I/O lines.
      0811 AND #$F8
      0813 STA $01      Select 64 kB RAM memory configuration.

      0815 LDX #$0C     Copy the invoking part to 300-312.
      0817 LDA $0830,X
      081A STA $012C,X
      081D DEX
      081E BPL $0817

      0820 LDX #$8B     Copy the main part to $DD64-$DDEE.
      0822 LDA $083C,X
      0825 STA $DD63,X
      0828 DEX
      0829 BNE $0822

      082B PLA          Restore original memory configuration.
      082C STA $01
      082E CLI          Enable interrupts.
      082F RTS          Return.
```

The user invokes the following part by issuing SYS 300. This part changes the memory configuration and jumps to the main part.

```
      012C LDA $01
      012E TAX          Store original memory configuration to X register.
      012F AND #$F8
      0131 ORA #$04
      0133 SEI          Disable interrupts.
      0134 STA $01      Select 64 kB RAM memory configuration.
      0136 JMP $DDA4    Jump to the main part.
```

The main part actually consists of two parts. It may be a bit complicated, and it might teach new tricks to you.

```
      DDA4 TXA
      DDA5 PHA          Push original memory configuration on stack.
      DDA6 LDA $FFFA
      DDA9 PHA
      DDAA LDA $FFFB
      DDAD PHA          Store the original values of $FFFA and $FFFB.
      DDAE LDA #$16
      DDB0 STA $FFFA    Set ($FFFA) to point to RTI.
      DDB3 LDA #$C0
      DDB5 STA $FFFB
      DDB8 JSR $DDDC    Swap the auxiliary routines in.
      DDBB JSR $C000    Disable NMI's and initialize CIA2.
      DDBE PLA
      DDBF STA $FFFB    Restore original values to $FFFA and $FFFB.
      DDC2 PLA
      DDC3 STA $FFFA
      DDC6 JSR $C01D    Print the message.
      DDC9 JSR $DDDC    Swap the auxiliary routines out.
      DDCC PLA
      DDCD TAY          Load original memory configuration to Y register.
      DDCE LDA #$00     Push desired stack register value on stack
      DDD0 PHA          (clear all flags, especially the I flag).
      DDD1 TSX
      DDD2 INC $0102,X  Increment the return address.
      DDD5 BNE $DDDA    (RTS preincrements it, but RTI does not.)
      DDD7 INC $0103,X
      DDDA STY $01      Restore original memory configuration.

      (The 6510 fetches the next instruction from $DDDC, which is now
      connected to the CIA2's register $C, the Serial Port register.
      The initialization routine wrote an RTI to it. The processor also
      reads from $DDDD as a side effect of the instruction fetch,
      thus re-enabling NMI's.)

      DDDC LDY #$3F     Subroutine: Swap the memory areas $C000-$C03F
      DDDE LDX $C000,Y              and $DD64-$DDA3 with each other.
      DDE1 LDA $DD64,Y
      DDE4 STA $C000,Y
      DDE7 TXA
```

```
DDE8 STA $DD64,Y
DDEB DEY
DDEC BPL $DDDE
DDEE RTS

C000 INC $01      Enable the I/O area.
C002 LDX #$81
C004 STX $DD0D    Enable Timer A interrupts of CIA2.
C007 LDX #$00
C009 STX $DD05
C00C INX
C00D STX $DD04    Prepare Timer A to count from 1 to 0.
C010 LDX #$DD
C012 STX $DD0E    Cause an interrupt.

(The instruction sets SP to output, makes Timer A to count
system clock pulses, forces the CIA to load the initial value
to the counter, selects one-shot counting and starts the timer.)

C015 LDX #$40

(The processor now jumps to the NMI handler ($C016), and
the SP register starts to act as a memory place.)

C017 STX $DD0C    Write an RTI to Serial Port register.
C01A DEC $01      Disable the I/O area.
C01C RTS          Return.

C01D LDA $01
C01F PHA
C020 ORA #$03     Enable I/O and ROMs.
C022 STA $01
C024 LDY #$0C     Print the message.
C026 LDA $C033,Y
C029 JSR $FFD2
C02C DEY
C02D BPL $C026
C02F PLA
C030 STA $01      Restore the 64 kB memory configuration.
C032 RTS

C033 "!DLROW ,OLLEH"

(The string is backwards in memory, since I don't want to
waste cycles in explicit comparisons. This method results in
more readable code than doing a forward loop with an index
value $100-(number of characters).)
```

This program is not excellent. It has the following bugs:

- The 6510's memory management lines P0 and P1 (LORAM and HIRAM, respectively) are assumed to be outputs. If you issued the command POKE0,PEEK(0)AND252, this program would not work. This could be easily corrected by setting the P0 and P1 lines to output in the beginning of the interfacing routine (300 - 312):

```
LDA $00
ORA #$02
STA $00
```

- The program does not restore the original state of the CIA2 Control Register A or Interrupt Control Register. It might be impossible to start using the Kernal's RS-232 routines after running this.

- If the user redirected output to cassette or RS-232, interrupts would be required. However, they are completely disabled.

- If a non-maskable interrupt occurs while the loader part is being executed, the program will screw up. This will happen also in the main part, if an NMI is issued after disabling ROMs and I/O in $0134 but before exchanging the contents of the memory places $C016 and $DD7A.

Freezer cartridges

There are many cartridges that let you to stop almost any program for "back-up" purposes. One of the most popular of these freezer cartridges is the Action Replay VI made by Datel Electronics back in 1989. The cartridge has 8 kilobytes RAM and 32 kilobytes ROM on board,

and it has a custom chip for fiddling with the C64 cartridge port lines -EXROM, -GAME, -IRQ, -NMI and BA.

If the -NMI line is not asserted (the NMI interrupts are enabled), all freezer cartridges should be able to halt any program. When the user presses the "freeze" button, the cartridges halt the processor by dropping the BA line low. Then they switch some of their own ROM to the $E000 - $FFFF block by selecting the UltiMax configuration with the -EXROM and -GAME lines. After this, they assert the -NMI line and release the BA line. After completing the current instruction, the processor will take the NMI interrupt and load the program counter from the vector at $FFFA, provided that the NMI line was not asserted already.

This approach is prone to many flaws. Firstly, if the processor is executing a write instruction when the program is being halted, and if the write occurred outside the area $0000 - $0FFF, the data would get lost, if the UltiMax configuration was asserted too early. This can be corrected to some extent by waiting at least two cycles after asserting the BA line, as the processor will not stop during write cycles. However, this is of no help if the processor has not gotten to the write stage yet.

Secondly, if the instruction being executed is outside the area $0000 - $0FFF, or if it accesses any data outside that area, the processor will fetch either wrong parameters or incorrect data, or both. If the instruction does not write anything, will only corrupt one processor register.

Thirdly, if the NMI interrupts are disabled, pressing the "freeze" button does not have any other immediate effect than leaving the UltiMax mode asserted, which makes any system RAM outside the area $0000 - $0FFF unavailable. It also forces the I/O area ($D000 - $DFFF) on. If the program has any instructions or data outside the lowmost four kilobytes, it will eventually jam, as that data will be something else than the program expects.

One might except that reading from open address space should return random bytes. But, in at least two C64's, the bytes read are mostly $BD, which is the opcode for LDA absolute,X. So, if the processor has a "good luck", it will happily execute only LDA $BDBD,X commands, and it might survive to the cartridge ROM area without jamming. Or it could eventually fetch a BRK and jump to the cartridge ROM via the IRQ/BRK vector at $FFFE. The Action Replay VI has the familiar autostart data in the beginning of both the ROML and ROMH blocks by default, and that data could be interpreted as sensible commands. The Action Replay VI was indeed able to freeze my test program, even though I had covered its -RESET, -IRQ and -NMI lines with a piece of tape, until I relocated the program to the first 4 kilobyte block.

Building an unbeatable freezer circuit

As you can see, it is totally impossible to design a freezer cartridge that freezes any program. If the program to be freezed has disabled the NMI interrupts, and if its code runs mostly at $0000 - $0FFF or $D000 - $DFFF, the computer will more probably hang than succeed in freezing the program.

However, it is possible to make some internal modifications to a C64, so that it can freeze literally any program. You need to expand your machine to 256 kilobytes following the documents on ftp.funet.fi in the /pub/cbm/hardware/256kB directory. It will let you to reset the computer so that all of the 64 kilobytes the previous program used, will remain intact. If you add a switch to one of the memory expansion controller's chip selection lines, the program being examined will have no way to screw the machine up, as the additional memory management registers will not be available.

A few enhancements to this circuit are required so that you can freeze the programs without losing the state of the I/O chips. You will also need to replace the Kernal ROM chip with your own code, if you do not want to lose the state of the A, X, P and S registers. Unfortunately this circuit will not preserve the state of the processor's Peripheral lines (its built-in I/O port mapped to the memory addresses 0 and 1), nor does it record the program counter (PC). I have a partial solution to the PC problem, though.

If you are interested in this project, contact me. I will design the additional hardware, and I will program the startup routines, but I certainly do not have the time to program all of the freezer software.

Most of the freezer software could be in RAM, so it would be very easy
to develop it, and you could even use existing tools by patching them
slightly.

==
FLD - Scrolling the screen
by Marek Klampar (klampar@elf.stuba.sk)

Scrolling the screen

[inspirated by Pasi Ojala article 'Opening the borders' from issue#6]

From Pasi 'Albert' Ojala's (po87553@cs.tut.fi or albert@cc.tut.fi) article:

> _Scrolling the screen_
>
> VIC begins to draw the screen from the first bad line. VIC will know what line is a bad line by comparing its scan line counter to the vertical scroll register : when they match, the next line is a bad line. If we change the vertical scroll register ($d011), the first bad line will move also. If we do this on every line, the line counter in VIC will never match with it and the drawing never starts (until it is allowed to do so).
>
> When we don't have to worry about bad lines, we have enough time to open the borders and do some other effects too. It is not necassary to change the vertical scroll on every line to get rid of the bad lines, just make sure that it never matches the line counter (or actually the least significant 3 bits).
>
> You can even scroll the bad lines independently and you have FLD - Flexible Line Distance. You just allow a bad line when it is time to display the next character row. With this you can bounce the lines or scroll a hires picture very fast down the screen.

(*** end of Albert's paragraph ***)

Well, everything important was written. I'm just adding this:

For moving hires picture replace ORA #$10 by ORA #$30.

For another FX try to replace part of irq routine begining with ORA #$10 by:

```
        ORA #$C0
        STA $D016,
remove  JSR CHOFS,
replace LDX OFSET
by      LDX #$ff
and enjoy =)
```

The demonstartion program for FLD application

```
;---------------------------------------
; Commodore Cracker 1993
;---------------------------------------
FROM    = $32
TO      = $FA
;---------------------------------------
      *= $C000
;---------------------------------------
INIT    LDA #0
      STA DIR     ; Direction
      LDA #$FF    ; Set garbage
      STA $3FFF
      LDA #FROM
      STA OFSET   ; Set ofset
      SEI         ; Disable interrupt
      LDA #$7F    ; Disable timer interrupt
      STA $DC0D
      LDA #1      ; Enable raster interrupt
      STA $D01A
      LDA #<IRQ   ; Set irq vector
      STA $0314
      LDA #>IRQ
      STA $0315
      LDA #0      ; To evoke our irq routine on 0th line
      STA $D012
      CLI         ; Enable interrupt
      RTS
;---------------------------------------
IRQ     LDX OFSET
```

```
L2    LDY $D012   ; Moving 1st bad line
L1    CPY $D012
      BEQ L1      ; Wait for begin of next line
      DEY         ; IY - bad line
      TYA
      AND #$07    ; Clear higher 5 bits
      ORA #$10    ; Set text mode
      STA $D011
      DEX
      BNE L2
      INC $D019   ; Acknowledge the raster interrupt
      JSR CHOFS
      JMP $EA31   ; Do standard irq routine
;---------------------------------------
OFSET .BYTE FROM
DIR   .BYTE 0
;---------------------------------------
CHOFS LDA DIR     ; Change OFSET of screen
      BNE UP
      INC OFSET   ; Down
      LDA OFSET
      CMP #TO
      BNE SKIP
      STA DIR
SKIP    RTS
;---------------------------------------
UP    DEC OFSET   ; Up
      LDA OFSET
      CMP #FROM
      BNE SKIP
      LDA #0
      STA DIR
      RTS
```

==========

Tech-tech - more resolution to vertical shift.
by Pasi 'Albert' Ojala (po87553@cs.tut.fi _or_ albert@cc.tut.fi)
Written on 16-May-91 Translation 02-Jun-92

(All timings are in PAL, principles will apply to NTSC too)

One time half of the demos had pictures waving horizontally on the width of the whole screen. This effect is named tech-tech, and the audience was puzzled. You can move the screen only eight pixels using the horizontal scroll register. This effect was done using character graphics. How exactly and is the same possible with sprites ?

Horizontal scroll register can move the screen by eight pixels. This isn't even nearly enough to produce a really stunning effect. You have to move the graphics itself, fortunately with a resolution of one character position (one byte) only, the rest can be done with the scroll register. During one scan line there is no time to move the actual data, you can only move a pointer. Changing the video matrix pointer won't help, because VIC (video interface controller) will fetch the character codes only at certain times, called bad lines. You can change the character set pointer instead, because VIC reads the data it displays directly from the character set memory.

Character set-implementation has its restrictions

Because horizontal movement is done by changing the character sets, the picture or text must be pure graphic and the character codes in the video matrix must be in a numerical order. The normal picture is in the first character memory and in the next one it is shifted one character position to the right. One video bank can hold only seven full character memories besides the video matrix. This limits the movement of the picture to 56 pixels. It is possible to get more movement if you use smaller picture or another video bank.

The shift is done so that on each scan line we update the horizontal scroll register ($D016) with the three lowest bits of the shift value. We use the other bits to select the right character set ($D018). In a tech-tech the shift value changes during the display of the whole picture, and the values are stored in a table. In addition to that, the shift values should be put into two tables, one for the horizontal scroll register and another for the character set select. This is necessary, because there is no time for extra calculations on a bad line.

Because we have to change the character set and x-scroll dynamically, we also need a raster routine to show a tech-tech. A raster routine is a routine which is synchronized to the electron beam. This eats up the processor time: the bigger the picture, the less time is left over for other activities. On other than bad lines you can do other funny things, like change the color of the background or border.

An example program

The demo program uses video bank 2, memory addesses $4000-7fff. The video matrix is in the beginning of the bank. Only inverted chars are used for the graphics, this way we have all eight character memories available and the maximum shift is 64 pixels. The area for the tech-tech in the video matrix is eight character rows high, but it has identical graphics on every line. This is why we use only 320 bytes from each character set.

You can use a joystick to control the movement of the tech-tech. The stick decreases or increases the shift add value in a resolution of a half pixel. When the shift reaches its highest/lowest value, the direction of the add is reversed. Just experiment with it.

You can do it on the screen, how about borders ?

Because you cannot get characters to the border, you might think that it is impossible to make a tech-tech effect in the borders. It takes so much time to change sprite x-coordinates, that you can change only some of them. There is time for five sprite moves, if you do not need to change the most significant (9th) bit of the x-coordinate. On the other hand, you could design the movements directly to the sprites and then just change the images, but then the movements would be constant.

However, there is one trick you can use to get all of the sprites on the screen, with variable movements and color bars etc. You do not change the x-coordinates, but the data itself on each scan line. In fact you change the sprite image pointers. There is multiple sprite pictures, where the graphics is shifted horizontally, just like the normal tech-tech charsets. Because of this, the sprites have to be placed side by side. No gaps are allowed. By changing the image pointers you can get the graphics to move horizontally on each line as you wish. With multicolor sprites you have to remember that one pixel corresponds to two bits - the movement is not so smooth.

Wait ! How come there is enough time to change the sprite pointers, when there is not time to change the coordinates ? This is another pointer trick. VIC reads the sprite image pointers from the end of the current video matrix, normally $07f8. You just have to change the video matrix pointer ($D018) to change all of the image pointers. This takes only eight cycles and there is plenty of time left for other effects on each scan line. If you use only one video bank, you can get the sprite picture to 16 different places. This allows also another kind of effects, just use your imagination.

Tech-tech demo program (PAL)

```
BANK=   $96     ; The value of the video bank register (CIA2) in the tech-area
ZP=     $FB     ; Zero page for indirect addressing
START=  $4400   ; Start of the charsets (we use inverted chars)
SCREEN= $4000   ; Position of the video matrix
SHIFTL= $CF00   ; x-shift, lowest 3 bits
SHIFTH= $CE00   ; x-shift, highest 3 bittid (multiplied with two)
POINTER= $033C  ; Pointer to shift-table
VALUE=  $033D   ; Shift now
SPEED=  $033E   ; Shift change

*= $C000  ; Start address..

INIT    SEI             ; Disable interrupts
        LDA #$7F
        STA $DC0D       ; Disable timer interrupt
        LDA #$81
        STA $D01A       ; Enable raster interrupt
        LDA #<IRQ
        STA $0314       ; Our own interrupt handler
        LDA #>IRQ
        STA $0315
        LDA #49         ; The interrupt to the line before the first bad line
        STA $D012
```

```
        LDA #$1B
        STA $D011       ; 9th bit of the raster compare

        LDY #0
        LDX #$40
        STX ZP+1
        STY ZP
        TYA
LOOP0   STA (ZP),Y      ; Clear the whole video bank ($4000-7FFF)
        INY
        BNE LOOP0
        INC ZP+1
        BPL LOOP0

        LDA #>START
        STA ZP+1
        LDA #$32        ; Character ROM to address space ($D000-)
        STA $01
LOOP1   TYA             ; (Y-register is zero initially)
        LSR
        LSR
        LSR
        TAX
        LDA TEXT,X      ; Which char to plot ?
        ASL             ; Source
        ASL
        ASL
        TAX             ; low byte to X
        LDA #$D0
        ADC #0          ; high byte (one bit) taken into account
        STA LOOP2+2 ; Self-modifying again..
LOOP2   LDA $D000,X
        STA (ZP),Y
        INX
        INY
        TXA
        AND #7
        BNE LOOP2       ; Copy one char
        CPY #0
        BNE LOOP1       ; Copy 32 chars (256 bytes)
        LDA #$37        ; Memory configuration back to normal
        STA $01

LOOP3   LDA START,Y       ; Copy the data to each charset, shifted by one
        STA START+2056,Y  ;  position to the right
        STA START+4112,Y
        STA START+6168,Y
        STA START+8224,Y
        STA START+10280,Y
        STA START+12336,Y
        STA START+14392,Y
        INY
        BNE LOOP3
        LDA #0          ; Clear the pointer, value and speed
        STA POINTER
        STA VALUE
        STA SPEED

LOOP4   TYA             ; (Y was zero)
        ORA #$80        ; Use the inverted chars
        STA SCREEN,Y    ; Set the character codes to video matrix
        STA SCREEN+40,Y
        STA SCREEN+80,Y
        STA SCREEN+120,Y
        STA SCREEN+160,Y
        STA SCREEN+200,Y
        STA SCREEN+240,Y
        STA SCREEN+280,Y
        LDA #239        ; leave the last line empty
        STA SCREEN+320,Y
        INY
        CPY #40
        BNE LOOP4       ; Loop until the whole area is filled
        CLI             ; Enable interrupts
        RTS

IRQ     LDA #BANK       ; Change the video bank, some timing
        STA $DD00
        NOP
        NOP
```

```
        LDY POINTER     ; Y-register will point to x-shift
        JMP BAD         ; next line is a bad line
LOOP5   NOP
LOOP6   LDA SHIFTL,Y    ; Do the shift
        STA $D016       ; 3 lowest bits
        LDA SHIFTH,Y
        STA $D018       ; another 3 bits
        NOP : NOP : NOP : NOP : NOP : NOP ; waste some time
        NOP : NOP : NOP : NOP : NOP : NOP
        NOP : NOP : NOP
        LDA $D012       ; check if it is time to stop
        CMP #$78
        BPL OVER
        INY   ; next position in table
        DEX
        BNE LOOP5       ; No bad line, loop
BAD     LDA SHIFTL,Y    ; This is a bad line, a bit more hurry
        STA $D016
        LDA SHIFTH,Y
        STA $D018
        INY
        LDX #7          ; New bad line coming up
        JMP LOOP6

OVER    LDA #$97        ; Video bank to 'normal'
        STA $DD00
        LDA #22         ; Same with the charset
        STA $D018
        LDA #8          ; and the horizontal scroll register
        STA $D016

        LDA $DC00       ; Let's check the joysticks
        AND $DC01
        TAX
        LDY SPEED
        AND #8          ; Turned right, add speed
        BNE EIP
        INY
        CPY #4          ; Don't store, too much speed
        BPL EIP
        STY SPEED
EIP     TXA
        AND #4          ; Turned left
        BNE ULOS
        DEY
        CPY #$FC        ; Too much ?
        BMI ULOS
        STY SPEED
ULOS    LDA VALUE       ; Add speed to value (signed)
        CLC
        ADC SPEED
        BPL OK
        LDA SPEED       ; Banged to the side ?
        EOR #$FF
        CLC
        ADC #1
        STA SPEED
        LDA VALUE
OK      STA VALUE
        LSR             ; Value is twice the shift
        TAX             ; Remember the shift
        AND #7          ; lowest 3 bits
        ORA #8          ; (screen 40 chars wide)
        LDY POINTER
        STA SHIFTL,Y
        TXA
        LSR
        LSR
        LSR             ; highest 3 bits too
        ASL             ;  multiplicated by two
        STA SHIFTH,Y
        DEC POINTER

        LDA #1          ; Ack the interrupt
        STA $D019
        JMP $EA31       ; The normal interrupt routine

TEXT    SCR "THIS IS TECH-TECH FOR C=64 BY ME" ; Test text
                        ; SCR converts to screen codes
```

Basic loader for the Tech-tech demo program (PAL)

```
1 S=49152
2 DEFFNH(C)=C-48+7*(C>64)
3 CH=0:READA$,A:PRINTA$:IFA$="END"THENPRINT"<clr>":SYS49152:END
4 FORF=0TO31:Q=FNH(ASC(MID$(A$,F*2+1)))*16+FNH(ASC(MID$(A$,F*2+2)))
5 CH=CH+Q:POKES,Q:S=S+1:NEXT:IFCH=ATHEN3
6 PRINT"CHECKSUM ERROR":END
100 DATA 78A97F8D0DDCA9818D1AD0A9AD8D1403A9C08D1503A9318D12D0A91B8D11D0A0, 3802
101 DATA 00A24086FC84FB9891FBC8D0FBE6FC10F7A94485FCA9328501984A4A4AAABD5E, 4749
102 DATA C10A0A0AAAA9D069008D4EC0BD00D091FBE8C88A2907D0F4C000D0DDA9378501, 4128
103 DATA B9004499084C99105499185C99206499286C99307499387CC8D0E5A9008D3C03, 3258
104 DATA 8D3D038D3E039809809900409928409950409784099A04099C84099F0409918, 3236
105 DATA 41A9EF994041C8C028D0DB5860A9968D00DDEAEAAC3C034CE1C0EAB900CF8D16, 4464
106 DATA D0B900CE8D18D0EAEAEAEAEAEAEAEAEAEAEAEAEAEAEAAD12D0C9781016C8CAD0, 5850
107 DATA D9B900CF8D16D0B900CE8D18D0C8A2074CBBC0A9978D00DDA9168D18D0A9088D, 4132
108 DATA 16D0AD00DC2D01DCAAAC3E032908D008C8C00410038C3E038A2904D00888C0FC, 3160
109 DATA 30038C3E03AD3D03186D3E03100EAD3E0349FF1869018D3E03AD3D038D3D034A, 2139
110 DATA AA29070908AC3C039900CF8A4A4A4A0A9900CECE3C03A9018D19D04C31EA1408, 2759
111 DATA 0913200913201405030820060F1220033D3634200219200D050F12, 652
200 DATA END,0
```

Uuencoded C64 executable version of the demo program (PAL)

```
begin 644 tech.64
M`0@-"`$`4[(T.3$U,@`F"`(`EJ5(*$,ILD.K-#BJ-ZPH0[$V-"D`7`@#`$-(J
MLC`ZAT$D+$$ZF4$D.HM!)+(B14Y$(J>9(CS7R,G4Q3X\P\S2/B(ZGC0Y,34R;
M.H``DP@$`(%&LC"D,S$Z4;*E2"C&*,HH020L1JPRJC$I*2FL,3:JI4@HQBC*1
M*$$D+$:L,JHR*2DI`+0(!0!#2+)#2*I1.I=3+%$Z4[)3JC$Z@CJ+0TBR0:<S?
M`,P(!@"9(D-(14-+4U5-($524D]2(CJ``!D)9`"#(#<X03DW1CA$,$1$0T$YN
M.#$X1#%!1#!!.4%$.$0Q-#`S03E#,#A$,34P,T$Y,S$X1#$R1#!!.3%".$0Q+
M,40P03`L(#,X,#(`9@EE`(,@,#!!,C0P.#9&0S@T1D(Y.#DQ1D)#.$0P1D)%G
M-D9#,3!&-T$Y-#0X-49#03DS,C@U,#$Y.#1!-$$T04%!0D0U12P@-#<T.0"S+
M"68`@R!#,3!!,$$P04%!03E$,#8Y,#`X1#1%0S!"1#`P1#`Y,49"13A#.#A!]
M,CDP-T0P1C1#,#`P1#!$1$$Y,S<X-3`Q+"`T,3(X```*9P"#($(Y,#`T-#DYE
M,#@T0SDY,3`U-#DY,3@U0SDY,C`V-#DY,C@V0SDY,S`W-#DY,S@W0T,X1#!%,
M-4$Y,#`X1#-#,#,L(#,R-3@`30IH`(,@.$0S1#`S.$0S13`S.3@P.3@P.3DPU
M,#0P.3DR.#0P.3DU,#0P.3DW.#0P.3E!,#0P.3E#.#0P.3E&,#0P.3DQ."P@?
M,S(S-@":"FD`@R`T,4$Y148Y.30P-#%#.$,P,CA$,$1"-3@V,$$Y.38X1#`P-
M1$1%045!04,S0S`S-$-%,4,P14%".3`P0T8X1#$V+"`T-#8T`.<*:@"#($0PN
M0CDP,$-%.$0Q.$0P14%%045!14%%045!14%%045!14%%045!14%%045!040Q,
M,D0P0SDW.#$P,39#.$-!1#`L(#4X-3``-`MK`(,@1#E".3`P0T8X1#$V1#!"Q
M.3`P0T4X1#$X1#!#.$$R,#<T0T)"0S!!.3DW.$0P,$1$03DQ-CA$,3A$,$$Y_
M,#@X1"P@-#$S,@"!"VP`@R`Q-D0P040P,$1#,D0P,41#04%!0S-%,#,R.3`XN
M1#`P.$,X0S`P-#$P,#,X0S-%,#,X03(Y,#1$,#`X.#A#,$9#+"`S,38P`,X+'
M;0"#(#,P,#,X0S-%,#-!1#-$,#,Q.#9$,T4P,S$P,$5!1#-%,#,T.49&,3@V7
M.3`Q.$0S13`S040S1#`S.$0S1#`S-$$L(#(Q,SD`&PQN`(,@04$R.3`W,#DP;
M.$%#,T,P,SDY,#!#1CA!-$$T031!,$$Y.3`P0T5#13-#,#-!.3`Q.$0Q.40PI
M-$,S,45!,30P."P@,C<U.0!G#&\`@R`P.3$S,C`P.3$S,C`Q-#`U,#,P.#)$_
M,30P-3`S,#@R,#`V,$8Q,C(P,#,S1#,V,S0R,#`R,3DR,#!$,#4P1C$R+"`V&
4-3(`<PS(`(,@14Y$+#``````HS!
``
end
```

ACE-128/64 PROGRAMMER'S REFERENCE GUIDE (version 0.9, for Release #10)
by Craig Bruce <csbruce@ccnga.uwaterloo.ca>

1. INTRODUCTION

ACE is a program for the Commodore 128 and Commodore 64 that provides a command shell environment that is similar to that of Unix. It is still in the development stage, but enough of it is complete to be useful. BTW, "ACE" means "Advanced Computing Environment" (well, advanced for the 128/64).

So what is ACE all about? Well, originally I tried a very ambitious project of writing a multitasking operating system for the 128. It got it partially working, but it was much too fragile and incomplete to be released. It was a white-elephant project. So, then then it came to me that I was aiming much too high. What I needed was a much simpler system, one that would give the type of programming interface and built-in features that I wanted, but one that was close enough to the Commodore Kernal that it would not require much of a programming effort to hack together a minimal implementation. And thus, there was ACE-128 Release #1. And I saw it was good.

What I wanted was a system that would be easier to program than the Commodore Kernal with all its weird and wonderful device types, non-existent memory management, and single-application design. The

first important feature of this environment was to be able to pass arguments from a command line to an application program by typing them on the command line of the shell. It is so annoying to load up a program in BASIC, run it, and have it ask you for filenames in some highly inconvenient way.

Another important system feature is to make near and far memory management part of the system, and to make far memory convenient to use for storing massively bulky data. And so it was. Also, we want to use custom device drivers. Commodore didn't really come through with the device drivers it provided. They are all REALLY SLOW. And so that was, also, although more custom device drivers are needed. We want to have the capability of making programs work together, rather than having programs that are totally incompatible. This functionality is still under construction. Programs will work together in this uni-tasking environment by allowing a program to execute another program as a sub-task, and then having control return to the calling program upon exit. Finally, we want some good quality applications and a powerful command shell. This is still being worked on and progress is coming slowly. Oh, almost forgot; we also want all programs to work on both the C64 and C128, and they do.

This documentation refers to ACE release #10, which has not been released yet (or even programmed). In fact, the current release is #9. Release #10 will be spruced up from the inside out to support the system interface described in this document. The current release is not being described, because some of its features are not the best they could possibly be. Note, however, that the basic features, like "open", "read", "dirread", and argv are not going to change. Note also that the version number of this "P.R.G." is 0.9. This is because this is the first incarnation of this document and, considering its nature, is bound to hold a large number of small errors.

2. SYSTEM INTERFACE

This section describes the interface between user programs and the ACE kernel. I am very careful throughout this interface specification about revealing any internal details that you do not strictly need to know. The interface with ACE is not specified in terms of absolute addresses; to aid in portability and extensibility, all interfaces are specified in terms of symbolic assembler labels. All of the ACE code is currently written for the Buddy-128 assembler. Also, because these interface absolute addresses are subject to change from version to version of the kernel, executables compiled for use with an old version of ACE may not work with a new version.

2.1. ZERO-PAGE VARIABLES

There are four zero-page variables used for passing arguments in most system calls. They are as follows:

SYMBOL	BYTES	DESCRIPTION
zp	2	zeropage pointer
zw	2	zeropage word
mp	4	memory pointer
syswork	16	system work area / arguments

The first two, "zp" and "zw" are used in most calls. They store simple 16-bit values; "zp" usually stores pointers to strings in RAM0 memory. The "mp" variable is 32-bits in length and is used exclusively for passing far memory pointers for use with the far memory routines. All three of these variables will remain unchanged inside of system call unless they will contain a return value. "syswork" is a 16-byte array used mainly when there are too many arguments for other variables to hold, and all non-input and non-output bytes of "syswork" are subject to change by the kernel. All input arguments placed in the "syswork" locations will be preserved unless otherwise indicated. [Note: In Release #9, "zp" took on the rolls of both "zp" and "mp"].

2.2. SYSTEM VARIABLES

There are several non-zeropage variables for storing system status and return values:

SYMBOL	BYTES	DESCRIPTION
errno	1	error number code returned by failed system calls
aceID	2	proof that user program is running on top of ACE
aceArgc	2	argument count for current process
aceArgv	2	argument vector address for current process

```
aceMemTop           2   highest address, plus one, that user prog can use
aceShellPath        2   ptr to storage for search path for executable programs
aceShellAlias       2   ptr to storage for shell command aliases
aceCurDirName       2   ptr to storage for the current directory name
aceExitData         2   ptr to storage for exit status from the last called prg
aceDirentBuffer <next>  storage for directory entries read from disk
aceDirentLength     -   really a constant: length in bytes of "aceDirentBuffer"
aceDirentBytes      4   bytes in file (usually inexact)
aceDirentDate       8   date of file in "YY:YY:MM:DD:HH:MM:SS:TW" format
aceDirentType       4   type of file in null-terminated string
aceDirentFlags      1   flags of file, "drwx*-et" format
aceDirentNameLen    1   length of name of file
aceDirentName      17   null-terminated name of file
```

ERRNO: "errno" is used to return error codes from system calls. When a system call ends in error, it sets the carry flag to "1", puts the error code in "errno", and returns to the user program, after undoing any system work completed at the time the error is encountered and aborting the operation. An error code number is stored in binary in the single-byte "errno" location. The symbolic names for the possible error codes are given in the next section. If no error occurs in a system call, the carry flag will be cleared on return from the call. Note that not all system calls can run into errors, so not all set the carry flag accordingly.

ID: "aceID" is a two-byte variable. Its purpose is to allow user programs to be sure that they are executing on top of ACE. The low byte (i.e., the first byte) must be equal to the symbolic constant "aceID1", and the high, "aceID2". This will allow ACE applications to inform idiot users that they cannot simply BOOT the program from BASIC, but rather they must run ACE first.

ARGC: "aceArgc" is a two-byte unsigned number. It gives the number of arguments passed to the application by the program (usually the command shell) that called the application. The first argument is always the name of the application program, so the count will always be at least one. Other arguments are optional.

ARGV: "aceArgv" is a two-byte RAM0 pointer. Pay attention. This pointer points to the first entry of an array of two-byte pointers which point to the null-terminated strings that are the arguments passed to the application program by the caller. (A null-terminated string is one that ends with a zero byte). To find the address of the N-th argument to an application, multiply N by two, add the "aceArgv" contents to that, and fetch the pointer from that address. In this scheme, the ever-present application name is the 0-th argument. The argv[argc] element of the argument vector will always contain a value of $0000, a null pointer.

MEM-TOP: "aceMemTop" is a two-byte RAM0 pointer. This points to one byte past the highest byte that the application program is allowed to use. All application programs are loaded into memory at address "aceAppAddress" (next section), and all memory between the end of the progam code and "aceMemTop" can be used for temporary variables, file buffers, etc. The main problem with this approach is that there are no guarantees about how much memory your application will get to play with. Many applications, such as simple file utilities, can simply use all available memory for a file buffer, but other programs, such as a file compressor, may have much greater demand for "near" memory. [Note: this variable has a different name in Release #9].

SHELL-PATH: "aceShellPath" is a two-byte RAM0 pointer. This points to the page of memory that stores the pathnames of directories to search through in order to find executable files. Each pathname is stored as a null-terminated string, and the list is terminated by an empty string (containing only the zero character). This is intended to be used by the shell program, or any other program that is so inclined, to examine or alter the search path. The search paths specified in the path page are global variables and are used by all programs that make the "exec" system call. This mechanism for reading/altering the executable search path may be changed in the future.

SHELL-ALIAS: "aceShellAlias" is a two-byte RAM0 pointer. This points to the page of memory that stores the aliases to be used with the command shell. An alias is a string that is substituted in the place of a command name on a shell command line whenever a certain command name comes up. For example, you might specify if the user enters the command "list a:" that the command name "list" be string-replaced with "cls;xls -l". Each alias is stored with the command name first, followed by an equals character ("="), followed by the string to substitute, followed by a zero. The alias list is terminated by an empty string. This

mechanism may be extended in the future to allow multiple pages of aliases, or may be changed completely.

CUR-DIR-NAME: "aceCurDirName" is a two-byte RAM0 pointer. It points to the null-terminated string indicating the current directory. The user is not supposed to modify this value, and the value will not always give a full pathname. The implementation of this feature may need to change in future versions of ACE.

ACE-EXIT-DATA: "aceExitData" is a two-byte RAM0 pointer. It points to the 256-byte buffer allocated for user programs to give detailed return information upon exiting back to their parent program. See the "exit" system call. User programs are allowed to read and write this storage. An example use of this feature would be a compiler program returning the line number and character position, and description of a compilation error to a text editor, so the editor can position the cursor and display the error message for user convenience. The implementation of this feature may need to change in future versions of ACE.

DIRENT-BUFFER: "aceDirentBuffer" is a buffer used for storing directory information read with the "dirread" system call, and is "aceDirentLength" bytes long. Only a single directory entry is (logically) read from disk at a time. The individual fields of a read directory entry are accessed by the fields described next. This field is also used for returning disk name information and the number of bytes free on a disk drive (see the "dirread" system call).

DIRENT-BYTES: "aceDirentBytes" is a four-byte (32-bit) unsigned field. As always, the bytes are addressed from least significant to most significant. This field gives the number of bytes in the file. Note that this value may not be exact, since Commodore decided to store sizes in disk blocks rather than bytes. For devices that report only block counts (i.e., every disk device currently supported), the number of bytes returned is the number of blocks multiplied by 254. This field, as well and the other dirent fields are absolute addresses, not offsets from aceDirentBuffer.

DIRENT-DATE: "aceDirentDate" is an eight-byte array of binary coded decimal values, stored from most significant digits to least significant. The first byte contains the BCD century, the second the year, and so on, and the last byte contains the number of tenths of seconds in its most significant nybble and a code for the day-of-week in its least significant nybble. Sunday has code 1, Monday 2, etc., Saturday 7, and a code of 0 means "unknown". This is the standard format for all dates used in ACE. This format is abstracted as "YY:YY:MM:DD:HH:MM:SS:TW". For disk devices that don't support dates, this field will be set to all zeroes, which can be conveniently interpreted as the NULL date, negative infinity, or as the time that J.C. was a seven-year-old boy.

DIRENT-TYPE: "aceDirentType" is a three-character (four-byte) null-terminated string. It indicates what type the file is, in lowercase PETSCII. Standard types such as "SEQ" and "PRG" will be returned, as well and other possibilities for custom device drivers.

DIRENT-FLAGS: "aceDirentFlags" is a one-byte field that is interpreted as consisting of eight independent one-bit fields. The abstract view of the fields is "drwx*-et". "d" means that the item is a subdirectory (otherwise it is a regular file), "r" means the item is readable, "w" means the item is writable, and "x" means the item is executable. The "x" option is really not supported currently. "*" means the item is improperly closed (a "splat" file in Commodore-DOS terminology). The "-" field is currently undefined. "e" means that the value given in the "aceDirentBytes" field is actually exact, and "t" means the file should be interpreted as being a "text" file (otherwise, its type is either binary or unknown). The bit fields are all booleans; a value of "1" means true, "0", false. The "d" bit occupies the 128-bit position, etc.

DIRENT-NAME-LEN: "aceDirentNameLen" is a one-byte number. It gives the number of characters in the filename. It is present for convenience.

DIRENT-NAME: "aceDirentName" is a 16-character (17-byte) null-terminated character string field. It gives the name of the file or directory or disk. Filenames used with ACE are limited to 16 characters.

2.3. SYSTEM CONSTANTS

There are several symbolic constants that are used with the ACE system interface:

SYMBOL	DESCRIPTION

```
------------------      ------------------------
aceAppAddress           the start address of applications
aceID1                  the id characters used to identify ACE applications
aceID2                  ...
aceID3                  ...
aceMemNull              the far memory type code used to indicate null ptrs
aceMemREU               far mem type code for Ram Expansion Unit memory
aceMemInternal          far mem type code for internal memory
aceMemRLREU             far mem type code for REU memory accessed thru RAMLink
aceMemRL                far mem type code for direct-access RAMLink memory
aceErrStopped           error code for syscall aborted by STOP key
aceErrTooManyFiles      err: too many files already opened to open another
aceErrFileOpen          err: don't know what this means
aceErrFileNotOpen       err: the given file descriptor is not actually open
aceErrFileNotFound      err: named file to open for reading does not exist
aceErrDeviceNotPresent  err: the specified physical device is not online
aceErrFileNotInput      err: file cannot be opened for reading
aceErrFileNotOutput     err: file cannot be opened for writing
aceErrMissingFilename   err: pathname component is the null string
aceErrIllegalDevice     err: the specified device cannot do what you want
aceErrWriteProtect      err: trying to write to a disk that is write-protected
aceErrFileExists        err: trying to open for writing file that exists
aceErrFileTypeMismatch  err: you specified the file type incorrectly
aceErrNoChannel         err: too many open files on disk drive to open another
aceErrInsufficientMemory err: ACE could not allocate the memory you requested
aceErrOpenDirectory     err: you are trying to open a dir as if it were a file
aceErrDiskOnlyOperation err: trying to perform disk-only op on char device
aceErrNullPointer       err: trying to dereference a null far pointer
aceErrInvalidFreeParms  err: bad call to "pagefree": misaligned/wrong size
aceErrFreeNotOwned      err: trying to free far memory you don't own
stdin                   file descriptor reserved for stdin input stream
stdout                  file descriptor reserved for stdout output stream
stderr                  file descriptor reserved for stderr output stream
```

"aceAppAddress", as discussed before, is the address that application programs are loaded into memory at. They must, of course, be compiled to execute starting at this address.

The "aceMem" group of constants are for use with the "pagealloc" system call, except for "aceMemNull", which may be used by application programs for indicating null far pointers. The "pagealloc" call allows you to specify what types of memory you are willing to accept. This is important because the difference types of memory have different performance characteristics. ACE will try to give you the fastest memory that is available. Ram Expansion Unit memory has startup and byte-transfer times of about 60 us (microseconds) and 1 us, respectively. This is the fastest type of far memory. Internal memory has a startup time of 24 us and a byte-transfer time of between 7 and 14 us (depending on whether accessing RAM0 or RAM1+). REU memory accessed through a RAMLink has a terrible startup time of 1000 us and a byte-transfer time of 2 us. Direct-access RAMLink memory has a startup time of 1000 us and a byte-transfer time of 16 us. All these times are for the C128 in 2 MHz mode.

The "aceErr" group gives the error codes returned by system calls. The error codes are returned in the "errno" variable. Not all possible error codes from Commodore disk drives are covered, but the important ones are. Finally, the "std" files group give the symbolic file descriptor identifiers of the default input, output, and error output file streams.

2.4. SYSTEM CALLS

All system calls are called by setting up arguments in specified processor registers and memory locations, executing a JSR to the system call address, and pulling the return values out of processor registers and memory locations.

2.1. FILE CALLS

```
NAME   :  open
PURPOSE:  open a file
ARGS   :  (zp) = pathname
          .A   = file mode ("r", "w", or "a")
RETURNS:  .A   = file descriptor number
          .CS  = error occurred flag
ALTERS :  .X, .Y, errno
```

Opens a file. The name of the file is given by a pointer to a null-terminated string, and may contain device names and pathnames as specified in the ACE user documentation. The file mode is a PETSCII

character. "r" means to open the file for reading, "w" means to open the file for writing, and "a" means to open the file for appending (writing, starting at the end of the file). An error will be returned if you attempt to open for reading or appending a file that does not exist, or if you attempt to open for writing a file that does already exist. If you wish to overwrite an existing file, you will have to call "remove" to delete the old version before opening the new version for writing.

The function returns a file descriptor number, which is a small unsigned integer that is used with other file calls to specify the file that has been opened. File descriptors numbered 0, 1, and 2 are used for stdin, stdout, and stderr, respectively. The file descriptor returned will be the minimum number that is not currently in use. These numbers are system-wide (rather than local to a process as in Unix), and this has some implications for I/O redirection (see the "fdswap" call below).

Restrictions: only so many Kernal files allowed to be open on a disk device, and there is a system maximum of open files. You will get a "too many files" error if you ever exceed this limit. Also, because of the nature of Commodore-DOS, there may be even tighter restrictions on the number of files that can be simultaneously open on a single disk device, resulting in a "no channel" error. Note that this call checks the status channel of Commodore disk drives on each open, so you don't have to (and should not anyway).

If the current program exits either by calling "exit" or simply by doing the last RTS, all files that were opened by the program and are still open will be automatically closed by the system before returning to the parent program.

```
NAME   :  close
PURPOSE:  close an open file
ARGS   :  .A   = File descriptor number
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Closes an open file. Not much to say about this one.

```
NAME   :  read
PURPOSE:  read data from an open file
ARGS   :  .X   = File descriptor number
          (zp) = pointer to buffer to store data into
          .AY  = maximum number of bytes to read
RETURNS:  (zw) = .AY=number of bytes actually read in
          .CS  = error occurred flag
          .ZS  = EOF reached flag
ALTERS :  .X, errno
```

Reads data from the current position of an open file. Up to the specified maximum number of bytes will be read. You should not give a maximum of zero bytes, or you may misinterpret an EOF (end of file). The buffer must be at least the size of the maximum number of bytes to read. The data are not interpreted in any way, so it is the programmer's responsibility to search for carriage return characters to locate lines of input, if he so desires. However, for the console the input is naturally divided up into lines, so each call will return an entire line of bytes if the buffer is large enough. There are no guarantees about the number of bytes that will be returned, except that it will be between 1 and the buffer size. So, if you wish to read a certain number of bytes, you may have to make multiple read calls.

The call returns the number of bytes read in both the .AY register pair and in (zw), for added convenience. [Note: in ACE Release #9, the number of bytes read are returned only in the .AY registers, not in (zw).] A return of zero bytes read means that the end of the file has been reached. An attempt to read beyond the end of file will simply give another EOF return. End of file is also returned in the .Z flag of the processor.

```
NAME   :  write
PURPOSE:  write data to an open file
ARGS   :  .X   = file descriptor number
          (zp) = pointer to data to be written
          .AY  = length of data to be written in bytes
RETURNS:  .CS  = error occurred
ALTERS :  .A, .X, .Y, errno
```

Writes data at the current position of an open file. [Note: ACE Release #9 also alters locations (zw) when writing to the console. This problem will be corrected.]

```
NAME   :  fastopen
PURPOSE:  open a file for fast reading
ARGS   :  (zp) = Name
          .A   = file mode (must be "r")
RETURNS:  .A   = file descriptor number
          .CS  = error occurred flag
ALTERS :  .X, .Y, errno

This performs the same function as the regular "open" call, except this
style of file accessing will allow files to be read much faster than the
other.  On devices that are so equipped, the "fastload" burst command is
used (similar shortcuts may be possible with other devices too).  The
drawback of this increased speed is that no other device I/O can take
place while a file is opened for "fast" access, not even to other
devices, except for output to the console.  Other files can be open,
just not accessed.  You also cannot open more than one "fast" file at a
time.  Interrupts will be disabled while a file is open for fast
accessing, so the user cannot re-enable them, for technical reasons.
The arguments to this call are exactly the same as the regular "open" to
make it easy to switch from using one to the other.  [Note: these "fast"
calls do not exist in Release #9].

NAME   :  fastclose
PURPOSE:  close the file that was opened for fast reading
ARGS   :  .A   = File descriptor number
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno

Closes the file that was opened for fast reading.

NAME   :  fastread
PURPOSE:  read data from the file opened for fast reading
ARGS   :  .X   = File descriptor number
          (zp) = pointer to buffer to store data into
          .AY  = maximum number of bytes to read
RETURNS:  (zw) = .AY=number of bytes actually read in
          .CS  = error occurred flag
          .ZS  = EOF reached flag
ALTERS :  .X, errno

Read data from the (one) file that is currently opened for "fast"
reading. The arguments and semantics are equivalent to the regular
"read" call.

NAME   :  bload
PURPOSE:  binary load
ARGS   :  (zp) = pathname
          .AY  = address to load file
          (zw) = highest address that file may occupy, plus one
RETURNS:  .AY  = end address of load, plus one
          .CS  = error occurred flag
ALTERS :  .X, errno

Binary-load a file directly into memory.  If the file will not fit into
the specified space, an error will be returned and the load truncated if
the device supports truncation; otherwise, important data may be
overwritten.

NAME   :  remove
PURPOSE:  delete a file
ARGS   :  (zp) = pathname
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno

Delete the named file.

NAME   :  rename
PURPOSE:  rename a file or directory
ARGS   :  (zp) = old filename
          (zw) = new filename
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno

Renames a file or directory.  If a file with the new name already
exists, then the operation will be aborted and a "file exists" error
will be returned.  On most devices, the file to be renamed must be in
the current directory and the new name may not include any path, just a
filename.

NAME   :  devinfo
```

```
PURPOSE:  give information about device
ARGS   :  .X   = file descriptor number
RETURNS:  .A   = device type code
          .X   = number of columns on device
          .Y   = number of rows per "page" of device
          .CS  = error occurred flag
ALTERS :  errno
```

This call returns information about the device of an open file. There are four possible values for the device type code: 0==console, 1==character-oriented device, and 2==disk device. The number of rows and columns per "page" of the device are also returned. For the console, this will be the current window size. For a character-oriented device, it will be the natural size (typically 80 columns by 66 rows), and for a disk, it will be 40 columns in 64 mode or 80 columns in 128 mode, both by 66 rows.

```
NAME   :  fdswap
PURPOSE:  swap two file descriptor numbers
ARGS   :  .X   = first file descriptor number
          .Y   = second file descriptor number
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This call swaps meanings of two file descriptor numbers. The file descriptors may be either in use or not use when the call is made. This call is intended to be used to redirect the stdin, stdout, and stderr file streams. To do this, simply open the new file intended to be, for example, stdout, and swap the file descriptor number returned from the open with file descriptor number 1 (stdout). Poof. Then call your subroutine or external program, and on return, swap the two file descriptors back, and close the redirection file.

2.2. DIRECTORY CALLS

```
NAME   :  diropen
PURPOSE:  open a directory for scanning its directory entries
ARGS   :  (zp) = directory pathname
RETURNS:  .A   = file descriptor number
          .CS  = error occurred flag
ALTERS :  .X, .Y, errno
```

This call opens a directory for reading its entries. It returns a "file" descriptor number to you to use for reading successive directory entires with the "dirread" call. The pathname that you give to this call must be a proper directory name like "a:" or "c:2//c64/games/:", ending with a colon character. You can have directories from multiple devices open for reading at one time, but you cannot have the directory of one device open multiple times. Also note that you cannot pass wildcards to this call; you will receive the entire directory listing.

```
NAME   :  dirclose
PURPOSE:  close a directory opened for scanning
ARGS   :  .A   = file descriptor number
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Closes a directory that is open for reading. You can make this call at any point while scanning a directory; you do not have to finish scanning an entire directory first.

```
NAME   :  dirread
PURPOSE:  read the next directory entry from an open directory
ARGS   :  .X   = file descriptor number
RETURNS:  .Z   = end of directory flag
          .CS  = error occurred flag
          aceDirentBuffer = new directory entry data
ALTERS :  .A, .X, .Y, errno
```

Reads the next directory entry from the specified open directory into the system interface global variable "aceDirentBuffer" described earlier. After opening a directory for reading, the first time you call this routine, you will receive the name of the disk (or directory). The "aceDirentNameLen" and "aceDirentName" fields are the only ones that will contain information; the rest of the fields should be ignored.

Each subsequent call to this routine will return the next directory entry in the directory. All of the "dirent" fields will be valid for these.

Then, after all directory entries have been read through, the last call

will return a directory entry with a null (zero-length) name. This corresponds to the "blocks free" line in a Commodore disk directory listing. The "aceDirentBytes" field for this last entry will be set to the number of bytes available for storage on the disk. On a Commodore disk drive, this will be the number of blocks free multiplied by 254. After reading this last entry, you should close the directory.

At any time, if something bizarre happens to the listing from the disk that is not considered an error (I don't actually know if this is possible or not), then the .Z flag will be set, indicating the abrupt ending of the directory listing.

```
NAME   :  isdir
PURPOSE:  determine whether the given pathname is for a file or a directory
ARGS   :  (zp) = pathname
RETURNS:  .A   = device identifier
          .X   = is a disk device flag
          .Y   = is a directory flag
          .CS  = error occurred flag
ALTERS :  errno
```

Given a properly formatted directoryname or filename, this routine will return whether the name is for a file or a directory, whether the device of the file or directory is a disk or character device, and the system identifier for the device. The two flags return $FF for true and $00 for false. The device identifier is superfluous for now, but a "devinfo" call may be added later. Note that this file does not indicate whether the file/directory actually exists or not.

```
NAME   :  chdir
PURPOSE:  change the current working directory
ARGS   :  (zp) = new directory pathname
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Changes the current working directory to the named directory. Too bad the Commodore Kernal doesn't have a similar call. Unlike the "cd" shell command, the argument has to be a properly formatted directory name. Note that only directories in native partitions on CMD devices are supported by this command; the 1581's crummy idea of partitions is not supported.

```
NAME   :  cdhome
PURPOSE:  change the current working directory back to the "home" directory
ARGS   :  <none>
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Changes the current working directory back to the "home" directory that is defined in the "config.sys" file as the initial directory.

```
NAME   :  mkdir
PURPOSE:  create a new directory
ARGS   :  (zp) = pathname of new directory
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Creates a new directory. I'm not sure, but I think that the current directory has to be the parent directory of the directory you want to create. This may be required by CMD devices, which will be the lowest common denominator for directory support. [Note: this call is not implemented in Release #9].

```
NAME   :  rmdir
PURPOSE:  delete an empty existing directory
ARGS   :  (zp) = pathname of empty directory to remove
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Deletes an existing directory. The directory must be empty (have no directory entries) in order for this command to succeed. Again, I am pretty sure that you have to be "in" the parent directory of the one to be deleted, since this is probably required by CMD devices. [Note: this call is not implemented in Release #9].

2.3. MEMORY CALLS

The calls given in this section are to be used for accessing "far" memory in ACE, which includes all REU, RAMLink, RAM1 and above, and sections of RAM0 that are not in the application program area. Applications are not allowed to access "far" memory directly, because

the practice of bypassing the operating system would undoubtedly lead to problems (can you say "MS-DOS"?).

All of these calls use a 32-bit pointer that is stored in the zero-page argument field "mp" (memory pointer). This field is to be interpreted as consisting of low and high words. The low word, which of course come first, is the offset into the memory "bank" that is contained in the high word. Users may assume that offsets within a bank are continuous, so operations like addition may be performed without fear on offsets, to access subfields of a structure, for example. You may not, however, make any interpretation of the bank word. An application should only access far memory that it has allocated for itself via the "pagealloc" call.

```
NAME   :  zpload
ARGS   :  [mp] = source far memory pointer
          .X   = destination zero-page address
          .Y   = transfer length
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Load zero-page locations with the contents of far memory. "mp", of course, gives the address of the first byte of far memory to be retrieved. The X register is loaded with the first address of the storage space for the data on zero page. It must be in the application zero-page space. The Y register holds the number of bytes to be transferred, which, considering that transfers must be to the application zero-page storage, must be 126 bytes or less. This routine will return a "reference through null pointer" if [mp] contains a null pointer.

```
NAME   :  zpstore
ARGS   :  .X   = source zero-page address
          [mp] = destination far memory pointer
          .Y   = transfer length
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This routine is the complement of "zpload"; this transfers data from zero page to far memory. The arguments and restrictions are the same as "zpload".

```
NAME   :  fetch
ARGS   :  [mp] = source far memory pointer
          (zp) = destination RAM0 pointer
          .AY  = transfer length
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This routine will fetch up to 64K of data from far memory into RAM0 memory where it can be accessed directly by the processor. The arguments should mostly speak for themselves. You should not fetch into RAM0 memory that is not specifically allocated to the application. You will get an error if you try to use a null far pointer.

```
NAME   :  stash
ARGS   :  (zp) = source RAM0 pointer
          [mp] = destination far memory pointer
          .AY  = transfer length
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This is the complement of "fetch" and operates analogously, except that it transfers data from RAM0 to far memory.

```
NAME   :  pagealloc
ARGS   :  .A   = requested number of pages to be allocated
          .X   = starting "type" of memory to search
          .Y   = ending "type" of memory to search, inclusive
RETURNS:  [mp] = far memory pointer to start of allocated memory
          .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This routine allocates a given number of contiguous far-memory pages for use by the application, and returns a pointer to the first byte of the first page. On calling, the accumulator contains the number of pages to allocate (a page is 256 contiguous bytes aligned on a 256-byte address (i.e., the low byte of a page address is all zeros)).

The X and Y registers contain the start and end "types" of far memory to search for the required allocation. The possible types are mentioned in

the System Constants section. The numeric values for the "aceMem" constants are arranged in order of accessing speed. So, if your application has speed requirements that dictate, for example, that RAMLink memory should not be used, then you would call "pagealloc" with a search range of .X=0 to .Y=aceMemInternal. If you wanted to say you are willing to accept any memory the system can give to you, you would specify .X=0 to .Y=255. The values of 0 and 255 will be converted to the fastest and slowest memory available. ACE will give you the fastest type of memory, from what you specify as acceptable, that it can. If you had an application that you didn't want to waste the high-speed memory on, you could first call "pagealloc" asking for slow memory, such as .X=aceMemRLREU to .Y=255, and if there is none of that type of memory left, make another call with .X=0 to .Y=aceMemRLREU-1.

This routine will then search its available free memory for a chunk fitting your specifications. If it cannot find one, the routine will return a "insufficient memory" error and a null pointer. Note that this error may occur if there is actually the correct amount of memory free but just not in a big enough contiguous chunk. If successful, this routine will return in "mp" a pointer to the first byte of the first page of the allocated memory.

If you call a subprogram with the "exec" call while the current program is holding far memory, that far memory will be kept allocated to your program and will be safe while the child program is executing. If you don't deallocate the memory with "pagefree" before exiting back to your parent program, then the system will automatically deallocate all memory allocated to you. So, have no fear about calling "exit" if you are in the middle of complicated far memory manipulation when a fatal error condition is discovered and you don't feel like figuring out what memory your program owns and deallocating it.

Some applications will want to have the most amount of memory to work with, and if there is free space in the application program area that the program is not using directly, then you may want to use that as "far" memory. To do this, you will need to write your own stub routines that manage page allocation and deallocation requests to the near memory, and calls the "pagealloc" and "pagefree" routines to manage the far memory. The "sort" program distributed with ACE does this. Please note that you CANNOT simply free the unused memory of the application program area and expect the system to manage it. Bad stuff would happen.

Some applications will want to have a byte-oriented memory allocation service rather than a page-oriented service. You can build a byte-oriented service on top of the page-oriented service in your application programs that manage memory for the application and ask the system for pages whenever more memory is required by the application. Note that this still means that allocated memory will be freed automatically when an application exits. The "sort" program implements this byte-oriented service, so you can check its source code to see how this is done (or to simply cut and paste the code into your own program).

```
NAME   :  pagefree
ARGS   :  [mp] = far memory pointer to start of memory to be freed
          .A   = number of pages to be freed
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This deallocates memory that was allocated to a process by using the "pagealloc" system call. You will get an error return if you try to deallocate memory that you don't own.

2.4. SCREEN CONTROL CALLS

This section describes the system calls that are available to application programmers for full-screen applications. These calls are intended to be general enough to handle different screen hardware (the VIC and VDC chips and a VIC soft-80-column bitmap screen, and possibly others). These calls are also designed to be efficient as possible, to discourage progammers from attempting to bypass using them. Bypassing these calls would be a bad thing.

The calls are designed around the C-128/PET concept of a window. There is only one active window on the display at a time, which may be is large as the entire screen or as small as 1x1 character cells. This window is very cheap to setup and tear down. An application can have multiple windows on the screen by switching the active window around.

In the calls below, all mention of "sw" in the arguments and return

values refer to the "syswork" array. For many calls, there is a "char/color/ high-attribute" argument. This argument determines which parts of a screen location will be modified. There are three components to each screen location: the character code, the color code, and the high-attributes. The character code is exactly the same as the PETSCII code for the character that you want to display (unlike the screen-code arrangement that Commodore chose). There are 128 individual characters in the normal PETSCII positions, and 128 reversed images of the characters in the most sensible other positions. The codes are as follows:

CODES (hex)	DESCRIPTION
$00-$1f	reverse lowercase letters
$20-$3f	digits and punctuation
$40-$5f	lowercase letters
$60-$7f	reverse graphics characters
$80-$9f	reverse uppercase letters
$a0-$bf	graphics characters
$c0-$df	uppercase letters
$e0-$ef	reverse digits and punctuation

There are sixteen color codes, occupying the lower four bits of the color value. These are RGBI codes, as follows:

CODE(dec)	(hex)	(bin) -rgbi	DESCRIPTION
0	$0	%0000	black
1	$1	%0001	dark grey
2	$2	%0010	blue
3	$3	%0011	light blue
4	$4	%0100	green
5	$5	%0101	light green
6	$6	%0110	dark cyan on VDC, medium grey on VIC-II
7	$7	%0111	cyan
8	$8	%1000	red
9	$9	%1001	light red
10	$a	%1010	purple
11	$b	%1011	light purple on VDC, orange on VIC-II
12	$c	%1100	brown
13	$d	%1101	yellow
14	$e	%1110	light grey
15	$f	%1111	white

Finally, there are the high-attribute bits. These occupy the four most significant bits of the color value. Depending on the type of display (VIC text, VDC text, or VIC/VDC bitmap), these bits have one of three meanings: character attributes, background character color, or no effect. Thus, care must be taken in using these bits; they will have different effects on different displays. The background character codes are the same as the foreground character codes listed above. The character attributes have the following meanings:

BIT VALUE -avub----	(dec)	(hex)	DESCRIPTION
%10000000	128	$80	alternate characterset (italic)
%01000000	64	$40	reverse character
%00100000	32	$20	underline
%00010000	16	$10	blink

These values are additive (or, should I say, "or-ative"); you can use any combination of them at one time. Normally, you may wish to leave the high-attribute bits alone, unless you take the values to give them from the color palettes (next section). To specify which of you wish to have changed, set bits in the "char/color/high-attribute" argument to system calls. The flags have the following values. They are or-ative as well:

BIT VALUE -cah-----	(dec)	(hex)	DESCRIPTION
%10000000	128	$80	modify character
%01000000	64	$40	modify color
%00100000	32	$20	modify high-attribute bits

The screen calls that deal with placing characters on the screen refer to screen locations using absolute addresses of locations in screen memory. This scheme is used for increased efficiency. You can obtain information about the absolute screen address of the top left-hand corner of the current window and the number of screen addresses between successive rows, to figure out screen addresses for your applications. For added convenience, there is a call which will accept row and column

numbers and return the corresponding absolute screen address.

The screen-control system calls are as follows:

```
NAME   :  winmax
ARGS   :  <none>
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Sets the current window to cover the entire screen.

```
NAME   :  winclear
ARGS   :  .A   = char/color/high-attribute modification flags
          .X   = character fill value
          .Y   = color fill value
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

This call "clears" the current window by filling it with the character/color you specify. You can use the char/color/hi-attr to limit what gets cleared. [Note: The arguments for this call are slightly different in Release #9].

```
NAME   :  winset
ARGS   :  .A   = number of rows in window
          .X   = number of columns in window
          sw+0 = absolute screen row of top left corner of window
          sw+1 = absolute screen column of top left corner of window
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

Sets the current window to the size you specify. You will get an error return if the window will not fit on the screen or of it does not contain at least one character. [Note: This call is not implemented in Release #9].

```
NAME   :  winsize
ARGS   :  <none>
RETURNS:  .A   = number of rows in window
          .X   = number of columns in window
          sw+0 = absolute screen row of top left corner of window
          sw+1 = absolute screen column of top left corner of window
         (sw+2)= screen address of top left corner
         (sw+4)= screen address increment between successive rows on screen
ALTERS :  <none>
```

Returns information about the current window. [Note: the arguments are slightly different in Release #9].

```
NAME   :  winput
ARGS   : (sw+0)= absolute screen address to start putting data at
         (sw+2)= character string pointer
          .X   = length of character string
          .Y   = color
          .A   = char/color/high-attribute modification flags
          sw+4 = fill character
          sw+5 = total field length
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Puts text onto the screen. The output region is given by the absolute starting screen address and the total field length. This region must be contained on one line of the current window, or bad things will happen. A pointer to the characters to be printed is given, as well as the length of the character array. Control characters in this string are ignored; they are poked literally onto the screen, including the null character. The length of the character string must be less than or equal to the total length of the field. Remaining spaces in the field will be filled in with the "fill character".

The color of the total field length will be filled in with "color". You can use the "char/color/hi-attr" modification flags to specify what is to be changed. If you were to, for example, specify that the colors of the field are not to be changed, then the call would execute faster.

```
NAME   :  wincolor
ARGS   :  .X   = new RGBI screen color
          .Y   = new RGBI border color
          .A   = which colors to change ($80=screen + $40=border)
RETURNS:  .X   = resulting RGBI screen color
          .Y   = resulting RGBI border color
```

```
ALTERS :  .A
```

Sets the color of the screen and border. You may optionally set one, the other, both, or neither. The resulting colors for colors changed, and the existing colors for colors unchaned will be returned. Note that not all screens have an adjustable color, so the border argument may be ignored.

```
NAME   :  winpos
ARGS   :  .A   = row
          .X   = column
RETURNS: (sw+0)= screen memory address of position
ALTERS :  .A, .X, .Y
```

Given a row and column in the current window, returns the corresponding absolute screen memory location for use with other calls. No errors are returned, so garbage in, garbage out.

```
NAME   :  wincursor
ARGS   : (sw+0)= screen address to place cursor
          .A   = enable flag ($ff=cursor-on / $00=cursor-off)
          .Y   = color to show cursor in
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Displays or undisplays the cursor at the given screen address. This call returns immediately in either case. No errors are returned. Do not display anything in or scroll the window while the cursor is being displayed, do not display the cursor twice, and do not undisplay the cursor twice in a row or bad things will happen. Also, make sure you give the same address when undisplaying the cursor as you did when displaying the cursor. When the system starts, the cursor will be in its undisplayed state (duh!). You also get to specify the color you want the cursor to be shown in. The high-attribute bits of this color are ignored.

```
NAME   :  winscroll
ARGS   :  .A   = flags: char/color/hi-attr + $08=up + $04=down
          .X   = number of rows to scroll up/down
          sw+4 = fill character
          .Y   = fill color
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Scrolls the contents of the current window up or down. You can scroll any number of rows at a time. After scrolling, the bottom (or top) rows will be filled with the fill character and color. You can limit whether the characters and/or colors are to be scrolled by using the "flags" byte in the usual way. Scrolling only the characters, for example, will be twice as fast as scrolling both characters and attributes. Whether to scroll up or down is specified also using bits in the "flags" field, as indicated in the input arguments above. You can specify scrolling in more than one way, and the result will be to scroll in each specified direction in turn, in the order up, then down. In the future, scrolling left and right may be added to this call. [Note: The arguments and semantics of this call are a little different in Release #9].

2.5. CONSOLE CALLS

The calls in this section refer to the system "console", which includes the screen and keyboard. The screen-related calls are at a higher level than the calls in the previous section.

```
NAME   :  stopkey
ARGS   :  <none>
RETURNS:  .CS  = stop key pressed
ALTERS :  .A, .X, .Y, errno
```

Indicates whether the STOP (RUN/STOP) key is currently being held down by the user. If so, carry flag is set on return (and clear if not). If the stop key is discovered to be pressed by this call, then the keyboard buffer will also be cleared.

```
NAME   :  getkey
ARGS   :  <none>
RETURNS:  .A   = keyboard character
ALTERS :  .X, .Y
```

Waits for the user to type a key (or takes a previous keystroke from the keyboard buffer). Regular characters are returned in their regular PETSCII codes, but there are many special control keystrokes. They are

not listed here (yet) because I haven't figured out what all of the special codes should be, but all 256 possible character values will be covered. Special codes like "page up", etc. should help in standardizing control keystrokes for applications. The key code is returned in the accumulator. No errors are possible.

```
NAME   :  concolor
ARGS   :  .A   = which colors to modify: $02=character + $01=cursor
                 + $80=modify high-attributes of colors
          .X   = new RGBI character color
          .Y   = new RGBI cursor color
RETURNS:  .X   = resulting character color
          .Y   = resulting cursor color
ALTERS :  .A
```

Sets the character and cursor colors to be used by the console for the "read" and "write" system calls that refer to files opened to the console device. You can use the flags argument to limit what gets changed. [Note: flags argument is slightly different in Release #9].

```
NAME   :  conpalette
ARGS   :  <none>
RETURNS:  sw+0 = main character color
          sw+1 = cursor color
          sw+2 = status character color
          sw+3 = separator character color
          sw+4 = highlight character color
          sw+5 = alert character color
          sw+6 = screen border color
          sw+7 = screen background color
ALTERS :  .A, .X, .Y
```

Returns the palette of colors that are recommended to be used in applications. These colors are chosen by the user in the system configuration, so they can be interpreted as being what the user wants and expects applications to use. A different selection is made by the user for each different screen type, and the palette returned will be for the screen type currently in use. The high-attribute bits of these colors are valid. Eight colors are included in the palette, and you may interpret their meaning according to the application. The suggested usages are given in the return arguments listed above.

```
NAME   :  conscreen
ARGS   :  .A   = number of text rows required, minimum
          .X   = number of text columns required, minimum
RETURNS:  .A   = number of text rows you get
          .X   = number of text columns you get
          .CS  = error occurred flag (requested size cannot be given)
ALTERS :  .Y, errno
```

This call selects an appropriate display device, screen, and layout for displaying text. You ask for the minimum number of rows and columns you require on the screen, and the call returns to you what you receive. If the system cannot match your minimum requirements, an error will be returned, and the current screen will be unchanged. The clock speed of the processor will be changed to match the screen selected, if appropriate.

```
NAME   :  conpos
ARGS   :  .A   = row
          .X   = column
RETURNS:  .CS  = error encountered flag
ALTERS :  .A, .X, .Y
```

This call will set the screen location that the next console "read" or "write" system call will operate from. If the "cursor" position is outside the boundaries of the current window on the screen, an error will be returned. [Note: this function is not implemented in Release #9].

2.6. PROCESS CONTROL CALLS

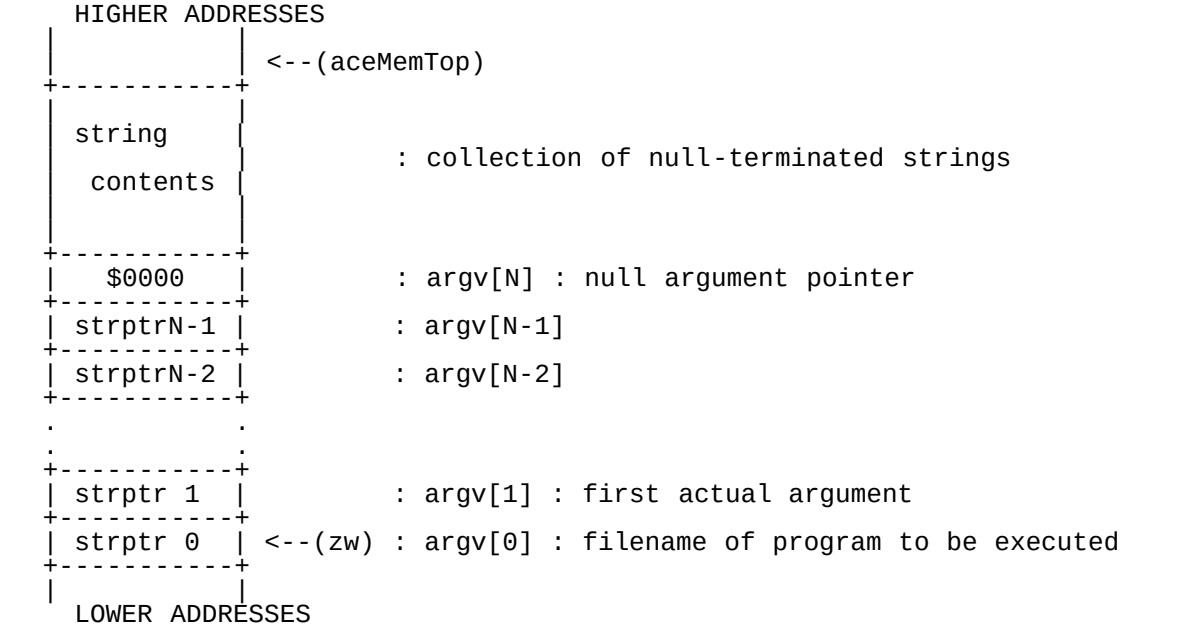
This section describes calls that are used to control the execution of processes (active programs). From within one program, you can call for the execution of another program, have it execute, and then return to the calling program. Since only one program is allowed in memory at a time, some special problems arise. Also, only rudimentary versions of these system calls are implemented in Release #9 and I haven't decided completely how they should work. So, this section is a bit tentative.

```
NAME   :  exec
```

```
PURPOSE:  execute external program as a child process
ARGS   :  (zp) = program name of executable
          (zw) = start address of argument vector
          .AY  = number of arguments
          [mp] = pointer to far memory volatile storage
RETURNS:  .A   = exit code
          .X   = number of bytes in "aceExitData" used
          [mp] = pointer to far memory volatile storage
          .CS  = error occurred flag
ALTERS :  .Y, errno
```

Calling this routine will cause a new "frame" to be set up on the "system stack" (lowering the available application area memory a little), the specified program to be loaded into memory over top of the current one, the new program to be executed, the old program to be reloaded from whatever disk unit it came from originally upon exit of the new program, and control to be returned to the old process with the return values from the executed program. This is a complicated procedure and many things can go wrong.

The first thing that a process that wants to call another program must do is set up the arguments to be passed in. All arguments must be null-terminated strings. These arguments are to be put into high memory, starting from one less than the location pointed to by "aceMemTop" and working downward. It does not matter in which order the strings are placed, as long as they are all grouped together. Then, immediately below the strings comes the vector of two-byte RAM0 pointers that point to the strings. This array must be in order, with the lowest entry pointing to the first (zero subscript) string, etc., the second highest entry pointing to the last string, and the highest entry containing the value $0000. An asciigram follows:



The first entry should indicate the filename or command name of the program being executed, and the subsequent arguments are the actual input arguments to the program being called. The address of the first argument vector table entry is loaded into (zw), and the number of arguments is loaded into .AY. Note that this value also includes the command name, so if, for example, you were to call program "wc" to count two filenames "hello" and "goodbye", then you would pass an argument count of 3. The name pointed to by "argv[0]" does not actually have to be the literal command name, but the one pointed to by (zp) does. If a relative executable name is given in (zp), then the search path will be used to locate the executable. Oh, don't screw up the organization of the arguments or bad things will happen; there is no structure checking.

After setting up the arguments, you'll want to set up any redirections of stdin, stdout, or stderr you'll be needing. Because there is only one open file table in the whole uni-tasking system, you'll have to manipulate existing entries using the "fdswap" system call described earlier. The open file table is inherited by the child process. Note that if it closes any of the open files it inherited, then they are also closed to your use also. If the child accidentally leaves open any files it opened, they will be closed by the system before you are reactivated.

Finally, before the call is made, you have to save any volatile local information into "far" memory. All application zeropage and application area memory will be modified by the called program, so you must save whatever you will need to continue after the return to be able to continue. As mentioned earlier, all of the "far" memory that a parent program owns will be safe, so you can save your volatile information there, in any format you wish. All you have to do is save the pointer to the far memory into the [mp] pointer. Upon return of the child process, the value you put into [mp] will be restored, and you can then restore your volatile information out of far storage. If you wish to save no volatile information, then you can just leave garbage in the [mp] value, since it will not be interpreted by the system.

Alright, so now you call the "exec" primitive, the child program is loaded, executed, and it returns.

At this time, the parent program (that's you) is reloaded from wherever it was loaded originally and you are returned to the instruction immediately following the "jsr exec", with your processor stack intact but the rest of your volatile storage invalid. Even if there is an error return (carry flag set), your volatile storage will still need to be restored, since the application area may have been overwritten before the error was discovered. In the case of an error return, the child process will not have been executed. If the system is unable to reload the parent program (you), then an error return is given to your parent, and so on, as far back as necessary. (This is a minor exception to the rule that an error return indicates that a child didn't execute; in this case, the child didn't complete).

You are also returned an "exit code", which will have application-specific meaning, although standard programs (e.g., shell script) interpret the value as: 0==normal exit, anything else==error exit. The X register is also set to indicate the amount of "aceExitData" that is used, to allow for more complicated return values.

[Note: This call is different in Release #9].

```
NAME   :  execsub
PURPOSE:  execute internal subroutine as a separate process
ARGS   :  (zp) = address of subroutine
          (zw) = address of argument vector
RETURNS:  .A   = exit code
          .X   = number of bytes in "aceExitData" used
          .CS  = error occurred flag
ALTERS :  .Y, errno
```

This call is very similar to "exec", except that it calls an internal subroutine rather than an external program. Thus, you don't have to save or restore your volatile storage, or worry about loading the child or reloading the parent. You do, however, set up the arguments and file redirections as you would for a full "exec". [Note: this call is different in Release #9].

```
NAME   :  exit
PURPOSE:  exit current program, return to parent
ARGS   :  .A   = exit code
          .X   = number of bytes in "aceExitData" used
RETURNS:  <there is no return, brah-ha-ha-ha-ha-ha!!!>
ALTERS :  <don't bloody well matter>
```

This call causes the current program to exit back to its parent. A program that exits simply by returning to its environment will give back an exit code of 0, which should be interpreted as a normal return. If you wish to indicate a special return, you should use some exit code other than zero. Many utilities will interpret non-zero error codes as actual errors and may abort further operations because of this.

You may set up a return data in "aceExitData", up to 255 bytes worth, and load the number of bytes used into .X if you wish. It is recommended that the first field of this data be a special identifier code so programs that cannot interpret your data will not try. You cannot give any far pointers in your return data, since all far memory allocated to you will be freed by the system before returning to your parent.

```
NAME   :  memstat
PURPOSE:  get "far" memory status plus process id
ARGS   :  <none>
RETURNS:  .A   = current process id
        [sw+0]= amount of "far" memory free
```

```
         [sw+4]= total amount of "far" memory
ALTERS :  .X, .Y
```

This call returns the current process id, the number of bytes of far memory currently free, and the total amount of far memory.

2.7. MISCELLANEOUS CALLS

```
NAME   :  utoa
PURPOSE:  convert unsigned 32-bit number to a decimal PETSCII string
ARGS   :  .A   = minimum length for return string
          .X   = zero-page address of 32-bit number
         (sw+0)= pointer to string buffer to store string
RETURNS:  .Y   = length of string
ALTERS :  .A, .X
```

This is a utility call in the kernel. It is really not necessary for it to be in the kernel, but so many programs make use of it that it makes sense for it to be factored out. You give a pointer to a 32-bit unsigned value in zero page memory, a pointer to a buffer to store that string that is at least as long as necessary to store the value plus the null-character terminator that will be put on the end of the string, and a minimum length value for the string. If the number requires fewer digits than the minimum length, the string will be padded with spaces on the left. Since a 32-bit quantity can only contain an maximum of ten decimal digits, the string buffer will only need to be a maximum of eleven bytes in size.

```
NAME   :  getdate
PURPOSE:  get the current date and time
ARGS   : (.AY) = address of buffer to put BCD-format date into
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Returns the current date and time in the BCD format described in the paragraph on "aceDirentDate". It puts it into the at-least-eight-byte storage area pointed to by (.AY).

```
NAME   :  setdate
PURPOSE:  set the current date and time
ARGS   : (.AY) = address of date in BCD format
RETURNS:  <none>
ALTERS :  .A, .X, .Y
```

Sets the current date and time in the system. (.AY) points to the BCD date string whose format is discussed in the paragraph on "aceDirentDate". No validity checking is performed on the date given.

```
NAME   :  cmdopen
PURPOSE:  open command channel to Commodore disk drives
ARGS   :  (zp) = device name
RETURNS:  .A   = file descriptor number
          .CS  = error occurred flag
ALTERS :  .X, .Y, errno
```

This "cmd" set of system calls really should not be present, but they will be needed until the full complement of disk-utility system calls are implemented. It is really not recommended that any application program rely on these calls being around very long. This call opens the command channel on the named device (standard ACE device name string) and returns the file descriptor number to use thereafter.

```
NAME   :  cmdclose
PURPOSE:  close command channel to Commodore disk drives
ARGS   :  .A   = file descriptor number
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This closes an opened command channel to a disk drive. Closing the status will NOT affect any other open files on the disk unit at the time.

```
NAME   :  cmdsend
PURPOSE:  send command over command channel to Commodore disk drives
ARGS   :  .X   = file descriptor number
         (.AY) = pointer to null-terminated command string
RETURNS:  .CS  = error occurred flag
ALTERS :  .A, .X, .Y, errno
```

This sends a command string to a disk drive. Since a null-terminated string representation is used, not all Commodore/CMD-DOS commands can be

sent, but the important ones can be.

```
NAME   :  cmdstatus
PURPOSE:  receive current status from command channel of Commodore disk drives
ARGS   :  .X   = file descriptor number
         (.AY) = pointer to buffer for null-terminated status string
RETURNS:  .A   = status code in binary
          .CS  = error occurred
ALTERS :  .X, .Y, errno
```

This returns the status of a disk drive in a string as well as the binary disk status number in the accumulator. The given status buffer must be at least 50 or so characters long (whatever is the longest possible disk status string).

3. USER PROGRAM ORGANIZATION

The ACE system itself is written using the Buddy-128 assembler, so it is recommended that applications be written in this also. User programs for ACE have a very simple structure. Here is the standard "hello, world" example program written in Buddy assembler for ACE:

```
-----=-----
.seq acehead.s
.org aceAppAddress
.obj "@0:hello"

jmp main
.byte aceID1,aceID2,aceID3

main = *
   lda #<helloMsg
   ldy #>helloMsg
   sta zp+0
   sty zp+1
   lda #<helloMsgEnd-helloMsg
   ldy #>helloMsgEnd-helloMsg
   ldx #stdout
   jsr write
   rts

helloMsg = *
   .asc "Hello, cruel world."
   .byte 13
helloMsgEnd = *
-----=-----
```

This would normally be put into a file called "hello.s". The ".s" extension means that this is an assembler file (a la Unix). The first thing this program does is include the "acehead.s" file. This is the Buddy assembler file that contains the header information declarations required to access the ACE system interface. The next line gives the start address to start assembling to; it must be "aceAppAddress", which is the address that ACE will load the program at. The next line is a directive to the assembler to write the executable code to a Commodore-DOS "PRG" file named "hello". This will be the command to enter at the ACE shell prompt.

The next six bytes of object code (which are the first six bytes of a program) describe the header required by ACE programs. The first three bytes must be a JMP to the main routine of the program. The next three bytes must have the values "aceID1", "aceID2", and "aceID3", respectively. And that's all there is to it. The rest of the program can be organized however you want it to be.

In this example, we set up the arguments for the "write" system call to print the string "Hello, cruel world." plus a carriage return to standard output. Note that this string does not need a terminating null ($00) character since the write call takes a buffer length. The program then returns to its calling environment via an RTS. This will cause an implied "exit(0)" to be performed by the system, returning to the parent program.

Although this program does not take advantage of this, an application program may use zero-page locations $0002 through $007f for storage without fear of having the storage trodden upon by the system. Also, the processor stack may be used from the point it was at upon entry to your program all the way down to the bottom. I will be doing something about ensuring there is always enough processor space for an application to use in the future, but for now, all applications have to share the single page of processor stack storage.

Finally, an application program starts at location "aceAppAddress" (plus six) and is allowed to use memory all the way up to one byte less than the address pointed to by "aceMemTop" for its own purposes. Currently, this amount of space is on the order of magnitude of about 24K. This will be increased in the future.

Application programs are not to access I/O features or even change the current memory configuration during execution. All I/O and unusual contortions must be performed by ACE system calls; otherwise, we could end up in as bad a shape as MESS-DOS.

4. CONCLUSION

Cool, eh?

==========

Looking Ahead: (Learned my lesson about "In The Next Issue" :-) (re: the mouse article etc....))

Either a Multi-Tasking article or a look at the Internals of Ace.

More graphics techniques.

Answers to the Trivia in this issue.

More articles and other information.

(Sorry that this is a little bit more vague than last time - Just have some authors actually finding real jobs, and others that forgot to include what they were going to do for the next issue.)